Atelier B

Interactive Prover

User Manual

version 3.7

CLEARSY
SYSTEM ENGINEERING

ATELIER B
Interactive Prover User Manual
version 3.7

Document made by CLEARSY.

CLEARSY
Maintenance ATELIER B
Parc de la Duranne
320 avenue Archimède
Les Pléiades III - Bât. A
13857 Aix-en-Provence Cedex 3
France

Tel 33 (0)4 42 37 12 99
Fax 33 (0)4 42 37 12 71
email : maintenance.atelierb@clearsy.com

Contents

Chapter 1

Introduction

The present Manual gives the method we advise for project proof operations using Atelier B. The basis of this method is to break down these proof operations into two phases:

Tuning up phase : quick analysis of proof obligations and modification of the B components when these obligations show errors

Formal proof phase : full formal proof of proof obligations. B components will be modified no further.

Indeed this method is not the sole possible one. This Manual can be used in two ways:

- You can read it before using Atelier B
- You can also use it when proving your project for a step-by-step follow-up of the corresponding steps.

Warning: The present document does not replace the Prover Reference Manual. It gives no analytical description for each command. For example, the user will not find the list of keywords to use with the *ApplyRule* command. This Manual is rather aimed at helping the user to understand *which* command to use and *why*. To read this Manual, a good understanding of the B language and a few notions of Atelier B are required. Proof is a difficult activity requiring a good information availability. We advise you to to install the following documents on every proof workstation:

- the **B Language Reference Manual**. It contains a definition for each mathematical symbol, essential properties and significant examples (additionally it gives the ASCII equivalent for each operator).
- the **B-Book chapters 1, 2 and 3**. It provides the logical construction of all mathematical notions used for proof.
- the **Interactive Prover Reference Manual**: it describes the proof command syntax.

- the **Interactive Prover User Manual** (the present text): it indicates the procedure to follow within the proof method framework we are advising.

Furthermore, the user may use the Interactive Prover “memo” card to rapidly access commands and the Proof Obligations User Manual which describes how to interpret each proof obligation.

Overview

Chapter 2 is an introduction to the mathematical formal proof as applied in Atelier B. If you are not familiar with formal mathematical demonstrations we recommend you read this short chapter.

Chapter 3 is a global presentation of Atelier B proof tools. It shows how to access these tools from the interface menus and explains what you can do to drive manually a demonstration. If you have never used proof tools you are strongly advised to read this chapter.

Chapters 4, 5 and 6 describe the method advised in this Manual to perform proof activities. Chapter 4 breaks the main proof down into two phases: the project tuning-up phase and the formal proof phase itself. It shows how to learn what the current phase is and when to change phase. Chapter 5 describes the tuning-up phase while chapter 6 describes the formal proof phase. In the course of a project development in B you will be able to follow in this Manual every step of your proof activities: you will then have adequate remarks on each situation.

Chapter 8 gathers case-studies. There you will see on examples some tricks used in interactive proof.

Chapter 2

The formal proof: Reminders

Formal methods are based on a principle: using mathematical notions to represent the behavior of computer software - this is why we talk of formal modeling. Mathematical notions are therefore the basic elements the user has available to build a model corresponding to his/her needs. The better knowledge he/she has of these notions, the better he/she will use the language. Using a formal language enables you to express demonstrable statements and a fair knowledge of these mathematical notions enables to efficiently conduct these demonstrations.

The B language is based upon set theory. This theory and all notions that ensue are built in the B-Book by J.R. Abrial, chapters 1, 2 et 3. If you are not familiar with these mathematical notions we strongly advise you to read these three chapters which will give you a *structured* knowledge of the subject. The B language manual gives you the definition of every symbol and its key properties in a dictionary-like format. But a knowledge of each separate symbol does not replace an understanding of mathematical concepts. It is therefore important to study how the theory these concepts stem from is built.

We will then expose the various mathematical notions in the very order these are built. Warning: this chapter is only a summary to facilitate the use of Atelier B. It is in no way a mathematical course as rigor is lacking. The following notions are set out in an intuitive and informal pattern, following the organization order of the B-Book.

2.1 Symbols

Mathematical writing is very rich in symbols not used in computing. For example, we use the implication $\Rightarrow$, the overload $\lhd\!\!-$, etc. These symbols - required for a synthetic writing of formulas - are not available on a computer keyboard. This is why they are represented by ASCII characters: e. g. $\Rightarrow$ is represented by `=>`.

It is better to use the symbolic notation in all documents in order to ease reading instead of the ASCII one. Here we will not use ASCII. ASCII conversion of each symbol is given in the B language reference manual, you will need it when working with Atelier B.

2.2 Formal reasoning

A *formal reasoning* consists in *demonstrating* a *statement* under a set of *hypotheses* using a set of *inference rules*.

For example, we want to demonstrate $8 > 0$ under the $8 > 5$ hypothesis. Our statement is then $8 > 0$ and the set of hypotheses is reduced to $8 > 5$. Let us assume that we have "forgotten everything", that is we wish to only use those rules and hypotheses explicitly stated. We assume only the two following inference rules:

$$\begin{array}{ll} \text{if } 5 > 0\text{, and if } 8 > 5\text{, then } 8 > 0 & \text{(rule 1)} \\ 5 > 0 \text{ is always true} & \text{(rule 2)} \end{array}$$

By applying rule 1 to prove $8 > 0$, as we suppose we know $8 > 5$ (this is our hypothesis), it remains only to prove $5 > 0$. Our new statement is therefore $5 > 0$. We now apply rule 2 stating that $5 > 0$ is always true. Application of this rule does not produce any new goal, proof is thus ended.

These notions of proof are very intuitive and natural. It is nonetheless useful to understand them using the elements just examined, that is:

- demonstration of a statement under certain hypotheses,
- collecting permitted inference rules.

By convention we represent our set of hypotheses by **HYP**. To indicate that we add a H hypothesis to this set, we will thus write **HYP**,H.

What occurs when one of the hypotheses we assume is always false ? Must we for example consider that $8 < 0$ is valid under the false $5 < 0$ hypothesis? Intuitively this leads to examine an impossible case. Its answer might seem a mere convention, but it is not the case. The global coherence theory forces us to consider that **Any statement is TRUE under false hypotheses**. We later will see examples where this necessity occurs. This apparently very abstract notion is often used when using the B language. In some cases, the creation of a B component proof obligations can and must produce contradictory proof obligations (refer section 5.8). These last are TRUE and part of the component proof.

2.3 Propositional calculus

A *logical statement* can be intuitively defined as a true or false affirmation. So "the house is white" is as a logical statement as the question "is this sentence true or false" has a meaning. Conversely "the house" is not a logical statement. A logical statement is named a *predicate*.

Let P and Q be two predicates. We specify the following notations:

- $P \ \wedge \ Q$ (P and Q)
- $P \ \Rightarrow \ Q$ (P implies Q)
- $\neg P$ (not P)

In formal proof, these notions are used by the following rules:

- to prove $P \ \wedge \ Q$ under the **HYP** hypothesis, it is sufficient to prove P under **HYP**, then to prove Q under the same hypothesis.
- to prove $P \ \Rightarrow \ Q$ under the **HYP** hypothesis, it is sufficient to prove Q under the **HYP** hypothesis plus the P hypothesis, that is, after our conventions: **HYP**,P. This is known as the *deduction* rule. It is also said that P "raises" in hypothesis level.
- to prove $\neg P$ under the **HYP** hypothesis, we have the following rule: if there exists a Q predicate such as we can prove both Q and $\neg P$ under the **HYP**,P hypothesis then $\neg P$ is proven. Intuitively, by assuming P we have reached a contradiction.

Let us note that if P is always false, then $P \ \Rightarrow \ Q$ is always true. This follows from the deduction rule and agrees with our remark on false hypotheses in § 2.2. To simplify the handling of predicates whose true or false status is known, let us introduce the following notations:

- **btrue** is the always true predicate;
- **bfalse** is the always false predicate.

We now have to introduce the last propositional notations, defined after the previous ones:

- $P \ \vee \ Q$ (P or Q) is defined as $\neg P \ \Rightarrow \ Q$.
- $P \ \Leftrightarrow \ Q$ (P equivalent to Q) is defined as $(P \ \Rightarrow \ Q) \ \wedge \ (Q \ \Rightarrow \ P)$.

Definition of "or" (disjunction) needs to be commented. It intuitively indicates the following: to is to say that P or Q is true is to say that if P is false then Q is bound to be true (translation of $\neg P \ \Rightarrow \ Q$). This definition is not symmetrical in P and Q in spite of the possibility to demonstrate that $\neg P \Rightarrow Q$ and $\neg Q \Rightarrow P$ are equivalent, that is $P \vee Q$ is identical to $Q \vee P$. On the other hand, definition of $P \vee Q$ is an example justifying our assertion that every goal is true under false hypotheses (refer section 2.2). Indeed, let us consider the proposition **btrue** $\vee\ Q$. To have the definition of *or* corresponding to the natural notion, we want this statement to be always true. That is:

$$\textbf{btrue} \vee Q \quad \Leftrightarrow \quad \textbf{btrue}$$

According to the $\vee$ symbol definition, this is written:

$$\begin{aligned}\textbf{btrue} \vee Q &\Leftrightarrow \neg(\textbf{btrue}) \Rightarrow Q\\ &\Leftrightarrow \textbf{bfalse} \Rightarrow Q\end{aligned}$$

It is thus necessary to consider that **bfalse** $\Rightarrow Q$ is always true.

The reader will find in the B-Book a list of the propositional operators key properties. We shall give here a few properties - less basic but selected for their importance when using Atelier B:

- $(\textbf{bfalse} \Rightarrow P) \Leftrightarrow \textbf{btrue}$
- $(\textbf{btrue} \Rightarrow P) \Leftrightarrow P$
- $(P \Rightarrow \textbf{btrue}) \Leftrightarrow \textbf{btrue}$
- $(P \Rightarrow \textbf{bfalse}) \Leftrightarrow \neg P$

2.4 Quantified predicates

In order to express the properties of our components written in B language, we will need additional concepts. For example, we might have to demonstrate a property on a loop index:

$$\textit{indice} \in 1 \mathinner{..} 10 \Rightarrow \textit{indice} < \mathsf{MAXINT}$$

Many operators still don't write this. First we need the notion of variable.

- **Variable**: any non predefined identifier is a variable, built according to several rules and using letters, digits and _[1].

In Atelier B and for implementation reasons, single letter variables are not allowed (these are *Jokers*, refer section 6.2.2). The concept of variable enables us to introduce a key notion, the *universally quantified predicate*. If v is a variable and P a predicate, we have the following construction:

- $\forall v.P$ (read for all v, P.)

[1] the specific syntactic rules defining a variable are stated in the B language reference manual [**?**]

We say that the P predicate is *quantified* by the *universal quantification* $\forall v$. We also say that the *range* of the v quantified variable is the P predicate. Let us examine a few examples of quantified predicates:

$$\forall xx.(xx \in \mathbb{N} \;\wedge\; xx < 10 \;\Rightarrow\; xx < 100)$$
$$\forall var.(var = 10 \;\Rightarrow\; var < 100)$$

Let us note that for typing reasons **all universally quantified predicates must follow this form:** $\forall v.(P \;\Rightarrow\; Q)$ [2].

An other key remark: **the quantified variable name is of no consequence**. We say that a quantified variable is a **dummy variable**. For example :

$\forall xx.(xx = 10 \;\Rightarrow\; xx < 100)$
is equivalent to
$\forall yy.(yy = 10 \;\Rightarrow\; yy < 100)$

The range of the x dummy variable in $\forall x.P$ is only the P predicate. Specifically variables bearing the same name can be used without conflict in other predicates. For example :

$$xx = 2000 \;\wedge$$
$$\forall xx.(xx = 10 \;\Rightarrow\; xx < 100)$$

This predicate indicates that the xx "external" variable is worth 2,000 and also that any number equal to 10 is smaller than 100. There is no confusion between the "external" variable occurrence and that of the dummy variable. Although correct, such writing leads to confusion and must be avoided.

Inference rules relating to universally quantified predicates are slightly more complex as they call on the notion of *non free* variable in an expression, a notion we shall not study in this chapter. The main rule - restricted to predicates of the $\forall x.(P \;\Rightarrow\; Q)$ form - is as follows:

- To demonstrate $\forall x.(P \;\Rightarrow\; Q)$ under the **HYP** hypotheses, if the x variable is not used in **HYP**, we only have to demonstrate Q under the **HYP**,P hypotheses.

This rule, called *generalization rule* means that to prove that Q is true for any x variable verifying P, we only need a x variable verifying P and to prove Q under these hypotheses. There is indeed a problem if the x variable has already been used with another meaning in the hypotheses: we then have to rewrite $\forall x.(P \;\Rightarrow\; Q)$ using an other variable. Such predicate rewriting call on the notion of *substitution* that will not be developed in this mathematical introduction.

[2] Additionally, the type controller in Atelier B expects quantified predicates under the syntactic form: $\forall v.(P \;\Rightarrow\; Q)$ where P is a predicate typing all introduced variables

Chapter 3

The Prover: an introduction

This part is aimed at B language and Atelier B users knowing the principles of proof but who have never used Atelier B Prover. This is a ”guided tour” to show where Atelier B functions are and where their documentation is located.

A summary of the major concepts :

proof is used to find errors
proving is not programming

proof can be partly automated
the Prover cannot demonstrate that an obligation is false
the rule database is the set of the Prover mathematical knowledge
proof mechanisms select rules to be used
the Automatic Prover designates the database and mechanisms
forces *Fast, 0, 1, 2, 3* are levels of mechanisms

a proof obligation has a status: *Proved, Unproved*
proof commands control the Prover in automatic as well as interactive mode
a PO has a level of automatic demonstration
a PO has an interactive demonstration

the overall tool structure: an Automatic Prover, commands, automatic or interactive control
control mode is selected through the start menu

global situation window displays POs list
commands are entered through control window

communication between the Interactive Prover and its interface is in line mode
using the interface buttons is equivalent to entering commands

directing the proof without adding non-validated knowledge
manual demonstration may use validated rules
user rules are non-validated rules

3.1 Practical approach

3.1.1 An example

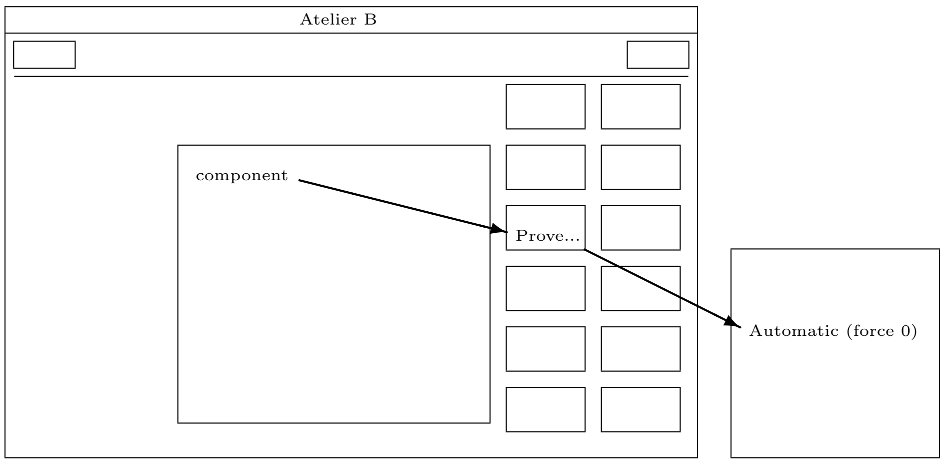
We first need an object to prove. You can take the following example.

```
MACHINE
    DemoExample
VARIABLES
    few, many
INVARIANT
    few ⊆ ℕ ∧
    many ⊆ ℕ ∧
    few ⊆ many
INITIALISATION
    few,many := {1,2,3},{2,3,4}
END
```

This component is purposely false: initialization does not establish the invariant as {1,2,3} is not included in {2,3,4}. It therefore produces a false proof obligation which allows us to find the error. We must never forget that proving is used to find errors in B sources. Nonetheless this example will allow us to make a tour of the proof tool. Let us also remember that proving is only used to validate software developed with B method and is not in any way a programming activity. It seems advisable to insist on this point as the user is often tempted to consider proofs as programs to be corrected and optimized - a state of mind leading to lengthy waste of time.

3.1.2 Automatic proof

Start the type-checker and the proof obligation generator associated with this component. Then launch the Automatic Prover (force 0):



Note the messages displayed in the start-up window. + indicate successful proofs while - failed ones. When the session ends, the Prover prints the final proof status:

```
Proving DemoExample

  Proof pass 0, still 3 unproved PO

    clause Initialisation
        ++-

End of Proof
  Initialisation        Proved 2        Unproved 1
TOTAL for DemoExample   Proved 2        Unproved 1
```

You have just launched the *Automatic Prover* under *force 0* (these terms will be explained later on). Out of the three proof obligations generated for this component, two have been automatically demonstrated and you no longer have to pay attention to these nor to what they verify. If all our component proof obligations had been demonstrated in the same way, nothing else would be needed: the component would be completely proved without further action of the user - cost of this proof phase being nil. But the user could request a written mathematical demonstration for each proof obligation to have the produced software certified but he/she would not do this for all obligations. To sum up, proving is at times fully automated.

In our example, one obligation is left undemonstrated. Then, either the component is true but the Prover has not found one of the demonstrations or the component is false and the remaining false obligation points to the error. Indeed we are in the second case as our component is intentionally false; the remaining false obligation is caused by $\{1, 2, 3\}$ not being included in $\{2, 3, 4\}$. Why does the Prover not clearly state that this demonstration is evidently false ? In fact, invalidating a proof obligation is a theoretically much more difficult task than demonstrating it, as *values* verifying the hypotheses must be selected - not the conclusions. This is why there is not at this moment an automated tool checking false proof obligations, notably when the Prover has not been designed to do so. We will then have to view this PO in order to check it is false. We will do this later.

How were the first two automatic proof obligations performed? We will now present the basic concepts in order to understand the Prover operation. Then we will be able to use it for interactive demonstrations.

- rules library: the set of rules making up the Prover mathematical knowledge. Roughly, these rules are instructions allowing the Prover to transform formulas. For example, a rule stating that any formula of the $a + b$ form can be replaced with $b + a$ (addition commutability).
- proof mechanism: Several rules can apply in a specific situation - their selection will have an influence upon the demonstration pattern. Let us resume with the previous example: we know that $a + b$ can be rewritten as $b + a$, but this does not tell us if the current demonstration will benefit from this transformation. Proof mechanisms are

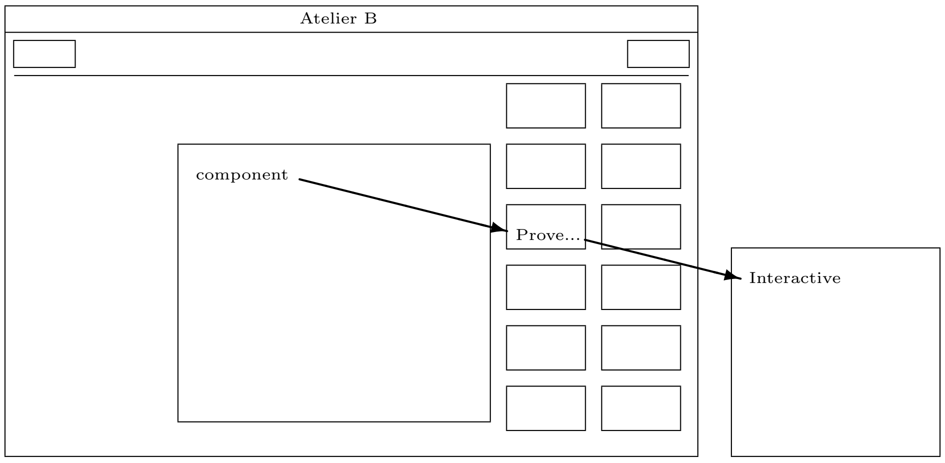
heuristic procedures enabling such choices. The mechanism of reduction of equalities is a significant example: when several variables are equal, this mechanism enables a minimum set of variables to express proof obligation.

- proof core: a set made up of the mathematical rules library and proof mechanisms. Proof obligations that have been directly successful are demonstrated by the proof core.
- force: Fast, 0, 1, 2, 3 : the Automatic Prover mechanisms are grouped in compatible sets called forces (refer section 4.2). The higher the force, the longer the proof - which can loop - but the more powerful it will be. Of all available forces, force 0 is the best compromise between performance and rapidity: it is the one to be used first.

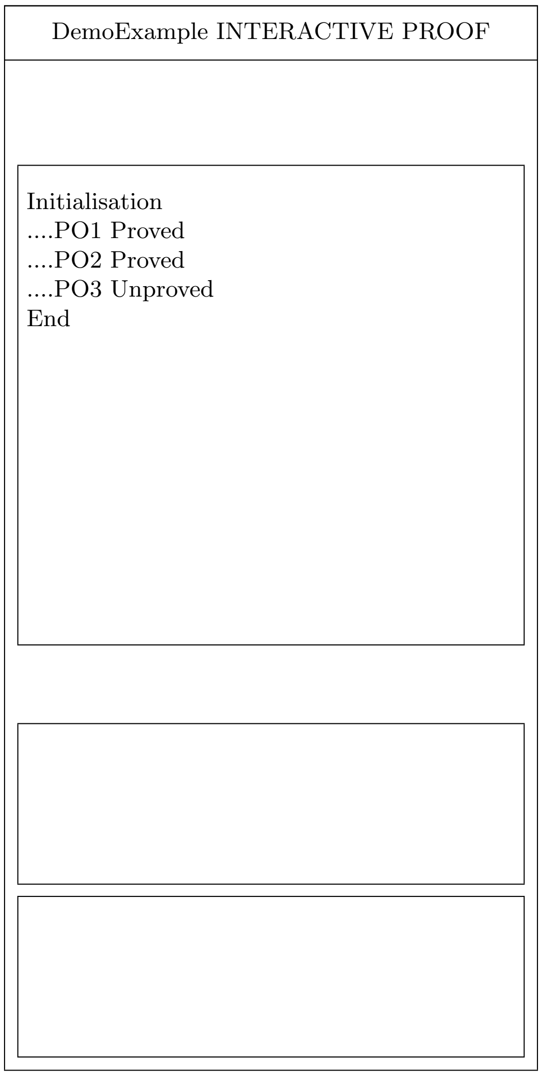
Previously, we launched on each proof obligation the Automatic Prover in force 0, that is using force 0 mechanisms. Successful demonstrations are only the application of rules extracted from the rule database as selected by force 0 mechanisms.

3.1.3 The interactive Prover

To study the remaining proof obligation, we are now entering the Interactive Prover. Select your component as specified below:



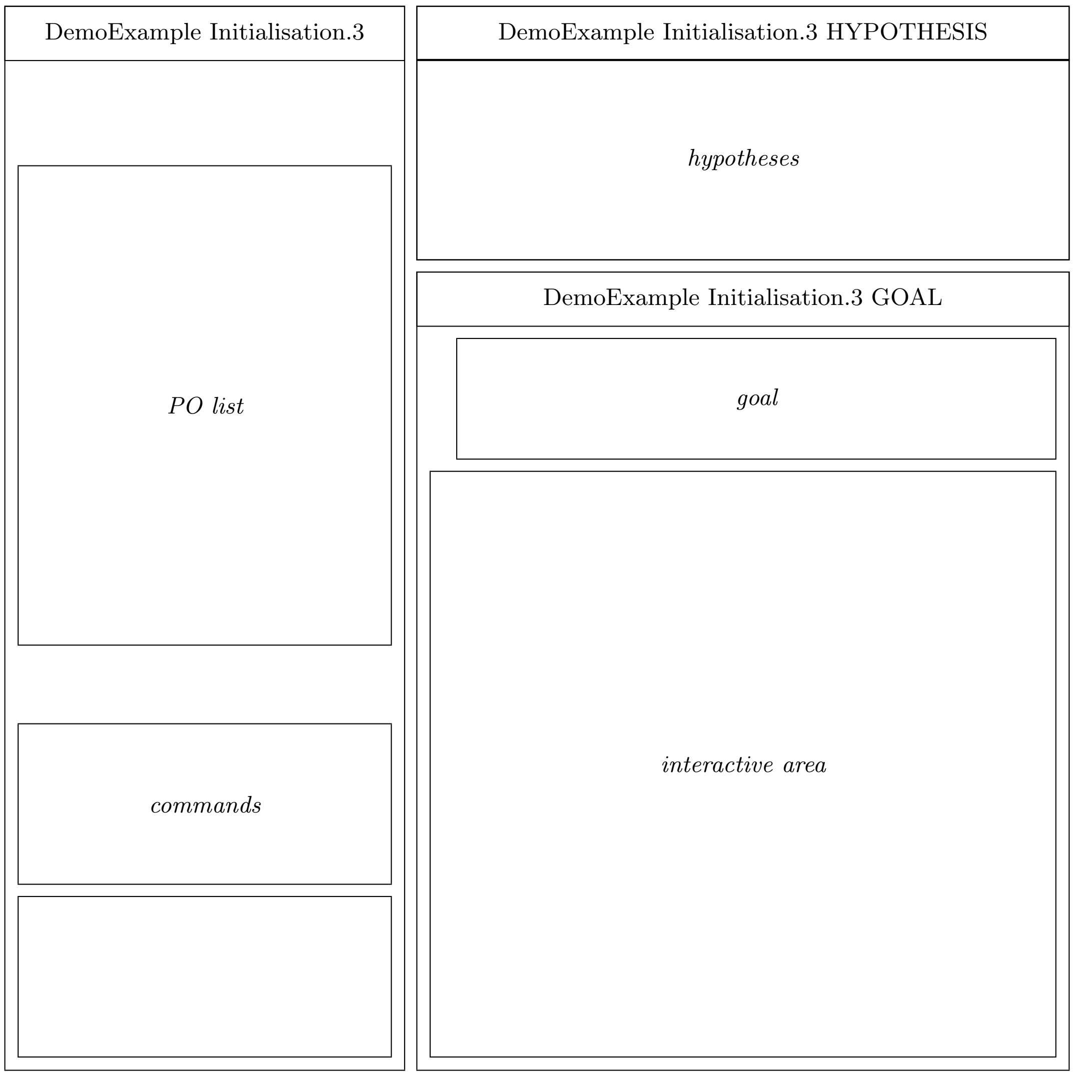
Atelier B main window is iconized and the Interactive Prover global situation window is displayed. The iconization of Atelier B main window can be understood from the importance of proof activities in a B project: the interface tries helping user concentration by displaying proof as an activity on its own. The displayed window is as follows:


DemoExample INTERACTIVE PROOF

Initialisation
....PO1 Proved
....PO2 Proved
....PO3 Unproved
End

This diagram shows key parts only: the title - “DemoExample INTERACTIVE PROOF” and the list of proof obligations (the various areas and buttons will be explained later on).A first remark: the *Proved* or *Unproved* status for each obligation is saved - this is crucial to know when proof is ended.

Click twice on “PO3 Unproved” to positionise the interactive proof on this proof obligation. Two additional windows are displayed and the screen is now as follows:


DemoExample Initialisation.3
DemoExample Initialisation.3 HYPOTHESIS
hypotheses
DemoExample Initialisation.3 GOAL
goal
PO list
interactive area
commands

In the goal area, at the end of the line we read: $1 \in \{2, 3, 4\}$ (the beginning of the line is a comment). This is the false goal that exposes the error, and in a rather analytic way. This is due to element 1 that $\{1, 2, 3\}$ is not included in $\{2, 3, 4\}$. Warning: if you have selected the ASCII default mode, goal $1 \in \{2, 3, 4\}$ is displayed as $1 : \{2, 3, 4\}$ because ":" is the ASCII symbol for membership. ASCII characters are required to enter formulae on a standard keyboard even when the interface can display mathematical fonts.

In the interactive area, enter `pr` and carriage return after the `PRI>` marker. Proof for this obligation then starts and fails on a goal which is always false: **bfalse**. With the `pr` command, you have launched the automatic Prover in standard force (force 0); this is similar to what we performed in automatic mode. The Automatic Prover is always available in interactive mode but it is not the only available command. There are other *proof commands* enabling to specifically apply a rule, to prove by cases, etc. All these commands are made up of two letters and are named in our documentation by a more explicit keyword. There is thus *DoCases* : `dc`, or *ApplyRule* : `ar`, and so on.

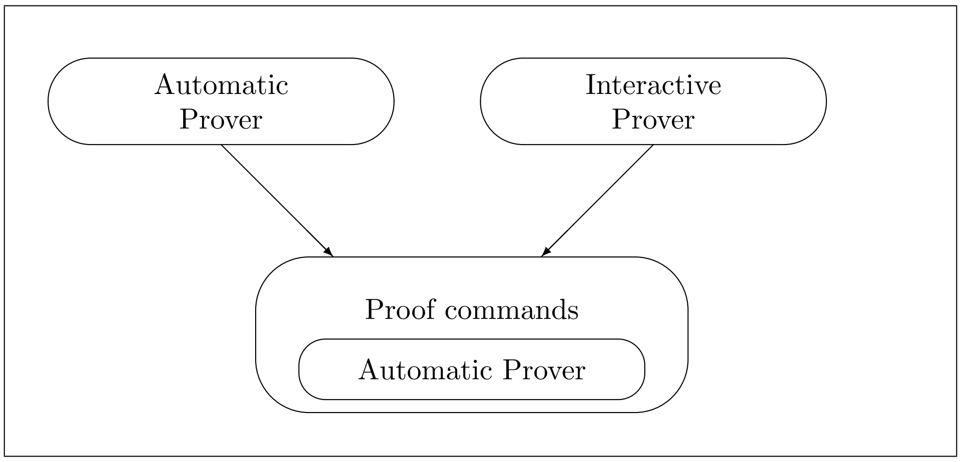
- proof commands: are commands controlling the proof. These can be calls to the

Automatic Prover (like command *Prove*, `pr`) or direct proof actions (such as apply a deduction *Deduction*, `dd`). Proof commands that each proof obligation can use are saved by the tool in a way we will later examine.

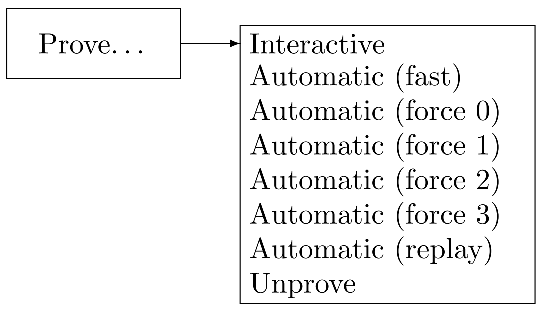
What we call the *Interactive Prover* is in fact a mode to control the automatic Prover where the *Prove* (`pr`) command is applied on every proof obligation. On the contrary, an interactive demonstration enables the user to select by himself/herself which proof commands are to be used to demonstrate an obligation. The series of selected commands is then saved together with the proof status - this is the *interactive demonstration.* In automatic mode, it is enough to save the maximum force for each obligation - for an obligation, this is the *automatic proof level.*

3.1.4 Overview of the proof tool

The overall operation of the proof tool (Automatic Prover and interactive Prover) can be understood on the following diagram:



When using the Interactive Prover, an interface enables you to send commands to the Prover. One of these is *Prove* (`pr`), that launches the Automatic Prover on current goal. You then have at your immediate disposal automatic proof mechanisms. When using the Automatic Prover, all component proof obligations are processed, either by applying a `pr` command or by replaying all recorded commands. In this case, these are the commands recorded during the last interactive session for all proof obligations. The global start menu of the proof tool is as follows:



Let us examine this menu:

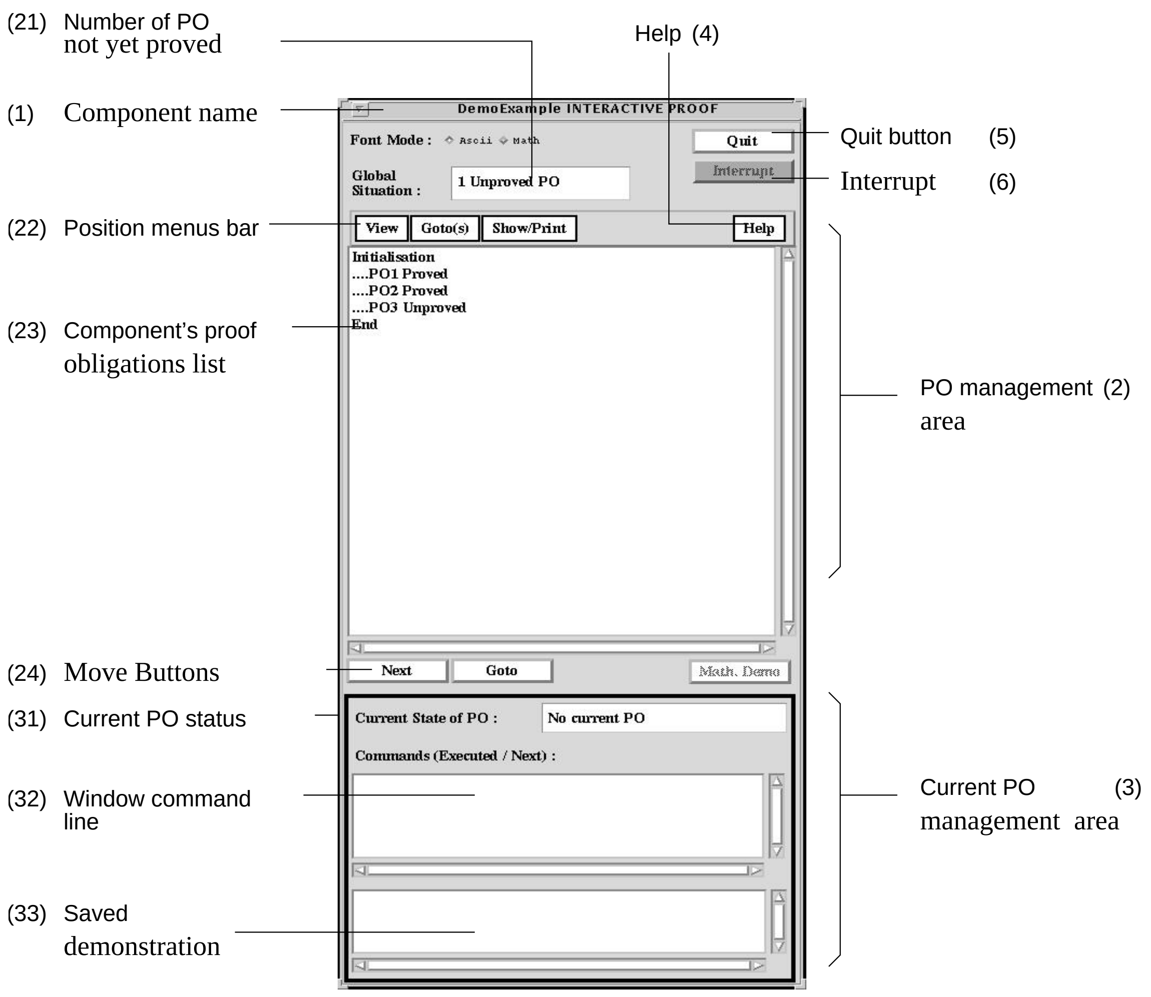
Interactive : launches the Prover in interactive mode. The various commands (moving between proof obligations, proof commands . . .) are entered by the user.

Automatic : launches the Prover in automatic mode on all the component unproved proof obligations. Using “force 0” to “force 3” options, the automatic Prover mechanisms are applied in each of the successive forces from 0 to 3 till the stated force. With the ”Fast” option only, the Fast option is used. With these options, the sole command in use is pr. But with the replay option, the sequence of interactively recorded commands for each lemma is replayed.

Unprove : resets all proof obligations of the component to “Unproved” status .

3.1.5 The main windows in detail

We will now describe the Interactive Prover global situation window, that displays the proof obligations list:



The window various areas are as follows:

1 The component name displayed in the window label bar.

2 The PO management area groups all items dealing with the component as a set.

> **21** The proof obligations meter displays the number of lemmas not yet proved. This area turns green when the component is entirely proved.
>
> **22** The position menus bar is used to set the display of the proof obligations list, to use the special *Goto* such as *GotoWithoutsave*, etc. The *Show/Print* button is used to print or save as a file the interactive proof elements.
>
> **23** The list gives the component proof obligations sorted by clausula, with their status. A double-click on a proof obligation is equivalent to a *Goto* on this proof obligation.
>
> **24** Move buttons are used to control position. *Next* moves to the next unproved proof obligation, *Goto* accesses a proof obligation specified in the list. *Mathematical Demo* writes in a file the proof obligation demonstration.

3 Current proof obligation management area groups all items specific to the current proof, that is the obligation specified by the previous *Goto*. It holds:

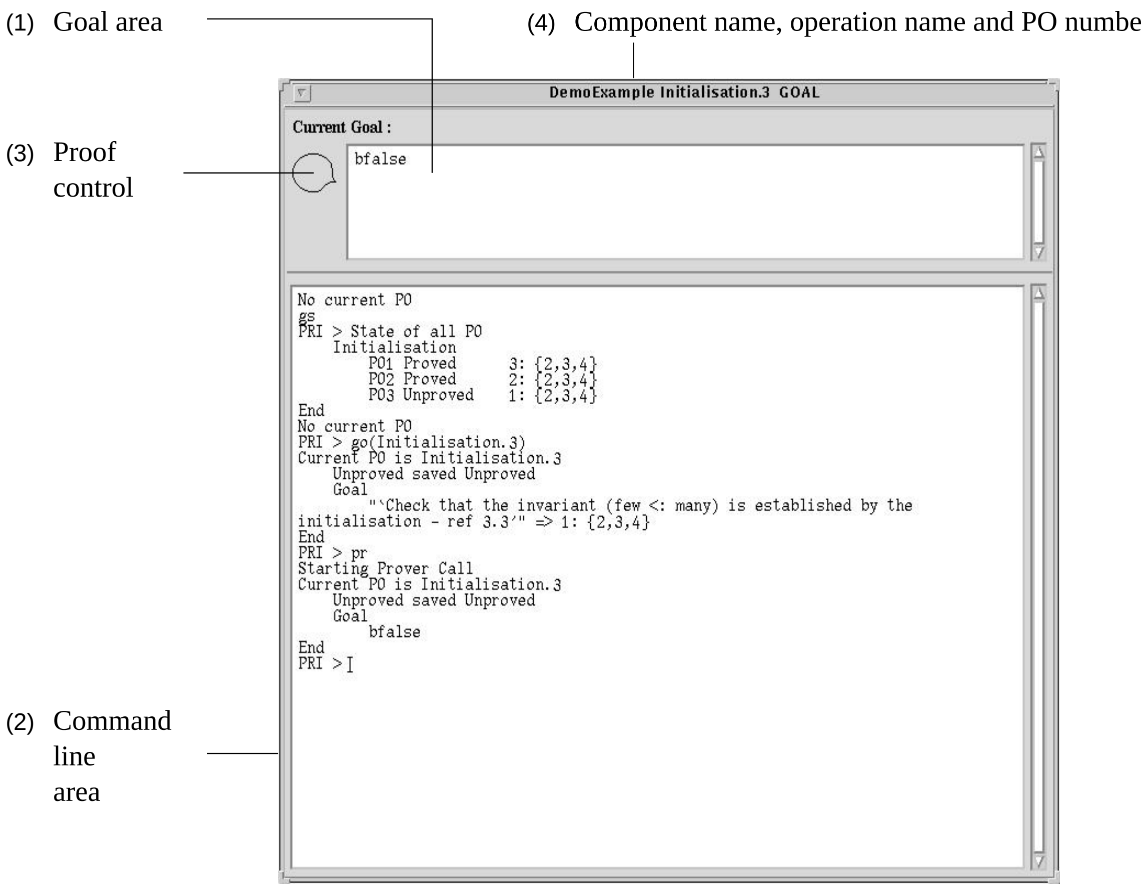
> **31** The current PO status, that is its status (Proved, Unproved) in the current demonstration and its status in the saved demonstration.
>
> **32** The window command line holds all proof commands performed on current PO, indented according to the proof tree.
>
> **33** The saved demonstration holds the command line saved for this proof obligation.

4 The Help button launches the Prover online documentation.

5 The Quit button stops the Prover.

6 The Interrupt button stops the last interactive proof command. It is used mostly to interrupt a looping *Prove* or *ApplyRule* command. This button is not accessible when the Prover is expecting a user command - as is the case on this image).

Let us examine the central window elements - a window that displays the goal and where we can enter the proof commands:


(1) Goal area
(4) Component name, operation name and PO numbe
(3) Proof control
(2) Command line area
DemoExample Initialisation.3 GOAL
Current Goal :

```
bfalse
```

```
No current PO
gs
PRI > State of all PO
    Initialisation
        PO1 Proved      3: {2,3,4}
        PO2 Proved      2: {2,3,4}
        PO3 Unproved    1: {2,3,4}
End
No current PO
PRI > go(Initialisation.3)
Current PO is Initialisation.3
    Unproved saved Unproved
    Goal
        "`Check that the invariant (few <: many) is established by the
initialisation - ref 3.3'" => 1: {2,3,4}
End
PRI > pr
Starting Prover Call
Current PO is Initialisation.3
    Unproved saved Unproved
    Goal
        bfalse
End
PRI >
```

This window elements are as follows:

1 The goal area: holds current goal displayed in a scroll-bar window. This area turns green when the demonstration is successful. The proof control then reads *Proved*.

2 the command line area where you enter all commands. There are four types of commands:

- Action commands: the proof commands as such. The most frequently used are:
  - *Prove* (`pr`): calling the Interactive Prover.
  - *AddHypothesis* (`ah`): adds an hypothesis, demonstrable using the current hypotheses.
  - *ApplyRule* (`ar`): direct use of a Prover rule or of an added one.
  - *DoCases* (`dc`): launches a proof by cases.
  - *useEqualityinHypothesis* (`eh`): uses an equality as hypothesis.
  - *SuggestforExist* (`se`): proposal for a goal of the form: $\exists x.P$.
  - *ParticularizeHypothesis* (`ph`): particularization of an hypothesis of the form: $\forall x.P$.
  - *FalseHypothesis* (`fh`): refutes a contradictory hypothesis.
- Position commands: these do not advance the proof but are used to move back or replay saved commands.
- Information commands: no action on the proof. They are used to search and display all information required to demonstrate a PO. For this type, key commands are *Search Hypothesis*, to search an hypothesis according to a specified filter and *Search Rule*, to search for a rule in a rules library.
- "Finalization" commands: demonstration generalization, interrupting a looping proof, requesting to stop a successful demonstration, etc.

All exchanges between Prover and its interface are displayed in this command line area. In the next section we will examine this key notion.

3 The proof control, displaying *Proved* when current demonstration is successful.

4 The window top title displays the component name, the name of the operation from which stems the proof obligation and the number of this obligation.

3.1.6 Exchanges with the interactive Prover

The proof tool is formed by two parts, the **interactive Prover** itself and its **man machine interface**.

The Prover performs commands (proof or information commands). The man / machine interface displays results and conveys your commands to the Prover. Dialog with the Prover is reduced to interactions of the type: commands for the tool / tool answers. This command mode interaction is fully displayed in the command line area. For example, when in the global situation window you press the *Next* button, the interface conveys a `ne` command (for *Next*) to the Prover just as if you had entered `ne` in the command line area. The Prover performs the command then it returns the current status as lines of text that the interface distributes among its windows - while leaving a trace in the answer, visible in the command line area.

So, the interactive proof interface always simulates a line mode dialog with the Prover, a dialog the user can directly hold from the command line area. All operations can be performed from this area (proof command, positioning, ...) but the syntax of each command must be known. Type `help` to obtain the list of available commands. These commands are always made up of two minus letters , the first two of the mnemonic make up the command.

For example:

The *Search Hypothesis* command is written `sh`

When the mnemonic is made up of one word only, the command takes its two first letters, as in `ne` for *Next* or `qu` for *Quit*. These commands are often taking up arguments such as `Goal`, `AllHyp` (use of these keywords could be avoided by using dialog buttons).

3.2 The interactive proof principle

How can commands previously discussed help an automatic proof succeed, where others failed ? How can you drive a proof to success using these commands? These are the questions answered in the present section. An example will illustrate such control.

We will demonstrate the following lemma:

$$\begin{array}{l} xx \in 1\,..\,10 \;\wedge \\ yy \in 2\,..\,10 \;\wedge \\ zz \in 3\,..\,10 \\ \Rightarrow \\ \mathsf{max}(\{xx, yy, zz\}) \leq 10 \end{array}$$

We assume that the interactive Prover mechanisms won't be up to demonstrate this. To demonstrate this lemma, we must make three cases according to the maximum, be it xx, yy ou zz. The user can start a first case by a *DoCases* command:

$$\mathtt{dc(}\mathsf{max}(\{xx, yy, zz\}) = xx\mathtt{)}$$

The proof then proceeds for $\mathsf{max}(\{xx, yy, zz\}) = xx$, then for $\mathsf{max}(\{xx, yy, zz\}) \neq xx$. It is possible that these two cases can be demonstrated by the Interactive Prover mechanisms - we shall assume this and in this case the proof is successful. **A user action launching a proof by case was sufficient to enable the demonstration**.

This example was aimed at making it understood that an interaction within a proof can bring it to success through the Interactive Prover mechanisms, without introducing any non-validated mathematical information. In fact you drive the proof by adding your intuition, then by launching the Prover again till a new failure or a success. To use an image from F. Meija, the user "plays billiards" with the Prover mechanisms.

It is evident that a good intuition of what the mechanisms will be up to is useful for this type of interactive proof. If we have to prove:

$$\begin{array}{l} xx \in \mathbb{N} \ \wedge \\ yy \in \mathbb{N} \ \wedge \\ yy \leq 10 \ \wedge \\ xx + 1 - 8 \leq yy \\ \Rightarrow \\ xx + 1 - 8 \leq 10 \end{array}$$

The automatic Prover mechanisms can fail on such a lemma as they first try to simplify the goal, that becomes $xx \leq 17$. It is then much more difficult to see the relation with the key hypothesis $xx + 1 - 8 \leq yy$. If the user sees this ill-selected simplification, he/she decides to act before calling the Automatic Prover (we will see the format of theses rules and how they are found using *SearchRule*). Let us assume that there is a rule: “OrderXY.77” that can demonstrate our PO. The command to apply it is *ApplyRule* - we will not describe its syntax nor our rule syntax here. The command entered would be, for example:

```
ar(OrderXY.77,Once)
```

The proof is successful. We are now dealing with a fully manually proof, without a call to the Automatic Prover and without adding any rule.

In some cases, it may be that the specific rule required for the proof is absent from the Prover database and that no other demonstration method is successful. The rule must then be added as a **manually rule**.

If manually rules have been used to prove a component, this rule can be false if some rules are false. It is then a rule outside the security bounds of Atelier B Prover - but validating this proof is simply validating user rules which is simpler than validating the proof itself. The number and complexity of these rules must then be small as compared to the size of the initial machine. Practically, this is done by occasionally using user rules together with the Prover security mechanisms. User rules are thus solved within sub-goals. User rules are written in the theorical language, in the `component.pmm` file. This file is fully written by the user, the tool does not create it by default so that the proof is not be validated if this file is missing.

To explain the interactive proof principle, we needed to review the rule concepts, rule application command, ... without any details. This will be done in chapter 6.

3.3 Conclusion

You now know how to use Atelier B Prover - and by what principles the proof obligations which automatic demonstration fails can be interactively demonstrated. The remaining part of this Manual insists more on proof methodology than on the description of various commands. A B project proof must indeed be conducted with a method.

Before using the Interactive Prover to demonstrate your project proof obligations, read chapter 4. By starting the formal proof directly you might waste a lot of time on non-key proof obligations and then discover errors whose correction invalidate previous proofs.

The Interactive Prover commands are fully described in the Prover Reference Manual. It is not necessary to read this reference completely in order to use the Prover, as the commands are described in chapter 6, in order of importance.Then you can consult the Reference Manual according to the needs.

Chapter 4

General method

4.1 Proof phases

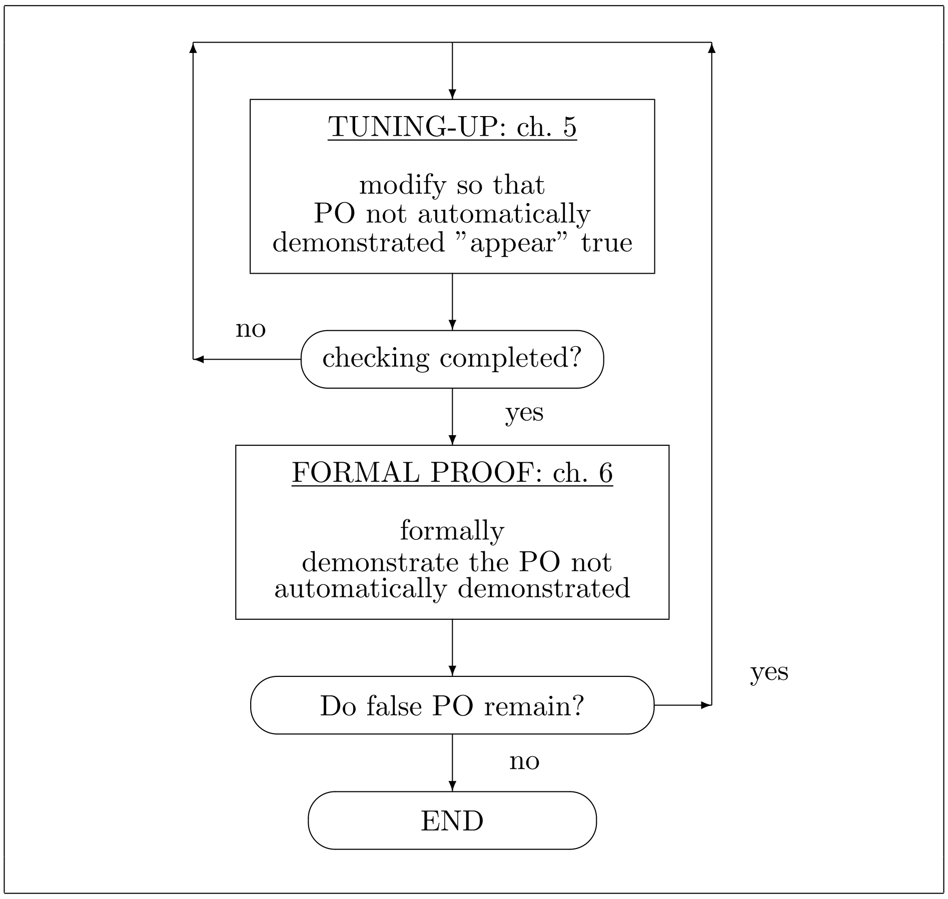
What are the proof activities involved when developing a software project using the B method and Atelier B? This point will be studied from an example: we assume a project made up of one sole abstract machine (specification) and of its implementation (the concrete program). The project will most certainly be built up in the following manner:

1. Write the abstract machine according to schedule;
2. Check need correct formalisation;
3. Launch the Automatic Prover on this abstract machine;
4. When not automatically demonstrated proof obligations remain, quickly check that they are true. If some are false, the abstract machine is non-coherent and must be corrected;
5. Write the implementation;
6. Read again this implementation in relation with the abstract machine;
7. Launch the Automatic prover on the implementation;
8. When undemonstrated proof obligations remain, check that these are true. If some are false, implementation is not correct and must be corrected;
9. Using the interactive Prover, demonstrate formally the remaining proof obligations in the abstract machine and in the implementation.

In the above development process, steps 3, 4, 7, 8 and 9 are proof steps. We see that the full formal proof is performed at the end: lengthy demonstrations must be avoided while components might have to be modified. This is why there are two separate phases in a B proof activity: components **tuning-up** by checking proof obligations and **final formal proof**.

This separation remains whatever the development method used. Note that in steps 3 and 7 we must use the interactive Prover in a fast configuration (force 0, refer next §) as we must wait for its completion to proceed to the following step.

It is crucial to know whether we are in a tuning-up phase or a final proof one. This phase method can be understood through the following diagram:


TUNING-UP: ch. 5
modify so that PO not automatically demonstrated "appear" true
no
checking completed?
yes
FORMAL PROOF: ch. 6
formally demonstrate the PO not automatically demonstrated
yes
Do false PO remain?
no
END

Warning: What we call "tuning-up" implicitly means tuning-up *from the proof point of view.* In this Manual, we will not deal with the global methods of writing and checking B language projects.

Must we complete the final tuning-up of all components before proceeding to the formal proof phase? Must we completely tune-up a component before writing the following one? This point will be developped later, as it depends on the project size and structure. All we can say is that you must not wait until you have written all the project components before tackling proof problems, and a component formal proof must not be started too early.

During formal proof phase, we assume we will not have to adapt the components, except when a proof obligation accepted as true is in fact false. **In this case, the impact of modifications on already performed demonstrations could produce delays**. This is why the tuning-up phase is so crucial.

Moving between proof phases is very tricky. During such moves, beware of the following traps:

- **Ensure that components do have their final form before the formal proof phase**. It is indeed commonplace to write components in a reduced or incomplete way and plan a finishing step. This must be done before the formal proof.
- **In a formal proof phase, ensure that all proof obligations can be assumed to be true.** Indeed, if a false proof obligation is discovered during the formal proof phase, the user is then tempted to proceed with this phase after modifying a component, whereas it is compulsory to make a new tuning-up phase.

4.2 Using the Prover forces

A user never expects a computer to design and create programs by itself - as the computer cannot guess what is to be obtained. On the contrary, what is to be obtained here is clear: we want demonstrations of statements to be proved using a set of known rules. Unfortunately, there is no algorithm that would produce the demonstration for any correct statement - automatic demonstrators, and therefore specifically Atelier B apply a set of more or less heuristic *strategies* that can fail or succeed. When we succeed a demonstration, it is correct but the failure of a strategy does not prove that the statement is false.

A major difference between proof and other more standard tasks such as program design is thus this possibility to succeed through the automated process of a computer. This is why it is always advisable to launch the automatic Prover on projects to be demonstrated whatever the required CPU time, in parallel with manual proof operations.

Strategies used in automatic proof are usually all the greedier in computing time as they are able to find complex demonstrations. Moreover, the most thorough strategies can induce infinite loops in the demonstrations. This is why the various strategies of Atelier B Prover are grouped in *force*. These various force are as follows:

Force	*Aproximate time per lemma*	*performance*
0	**always less than 10 seconds**	70%
1	from a few seconds to 2 or 3 minutes	+1%
2	from a few minutes to several times ten minutes	+3%
3	from several times ten minutes to several hours	+1%
“Fast”	less than 3 seconds	30%

Times stated above are a mere rating, they mainly relate to the first proof obligations in each operation. The following proof obligations have indeed many hypotheses in common with the first ones and this hypotheses process is factored. Performances are just a rating and given in percentage of obligations demonstrated in a full “standard” project. Performances of forces 1, 2 and 3 are specified as a gain according to the previous one because force 1, 2 and 3 are always used in a sequence starting with force 0. Thus the higher forces can only deal with lemmas not demonstrated in inferior force - this saves CPU time and limits the danger of inducing infinite loops. The ”Fast” force is used singly.

Force 0 is considered as the optimum balance between efficiency and computing time. This is the force to use to try to demonstrate proof obligations even before reading these, in order to restrict their number. These are indeed very numerous - a mean value is a proof obligation per executable code line produced. The ”Fast” force does not have sufficient performances for this task. Forces 1, 2 and 3 are used in parallel during tuning-up and formal proof phases in the hope that some proof obligations will be automatically demonstrated before having been manually processed.

The Atelier B Prover forces are used according to the following principles:

- **Use force 0**: never examine a proof obligation before attempting to prove it with the Automatic Prover in force 0.
- **“Use” your computers**: if you have at your disposal idle computers on which Atelier B has been installed, it is always useful to launch on these the Automatic Prover in force 1, 2 or 3 to demonstrate your project true proof obligations.
- **Do not wait**: do not wait for the Automatic Prover in force 1, 2 or 3 to have completed processing your project to start tuning-up or formal proof phases.

The Automatic Prover is also used for interactive proof. This might seem a paradox, but what we call an interactive proof is in fact semi-automatic, where user actions come in between calls to the Automatic Prover. We shall then have to select a force used also for interactive proof, it conditions all operations of the Automatic Prover in manual demonstrations. Most of the time, it is advised to use force 0; force 1 is also used sometimes. We will examine this point in chapter 6.

4.3 Well definition lemmas

It is possible to write expressions that look like mathematical expressions but have no meaning: for example $\mathsf{max}(\varnothing)$. We wrongly use the max operator that is defined only for at least a non-empty set of integers. Such expressions can cause problem when using automatic proof with inference rules. We will only approach these problems from their practical consequences. The **mdelta** tool allows to check *a posteriori* that a B project doesn't contain ill-defined expressions (refer to the Mdelta Tool User Manual) This check is perfomed by generating well definitions lemma. Most of the time, these well definition lemmas are obvious, it is enough to read them quickly to check them. If one of these lemmas is false then:

- Either there are ill-typed expressions within the component under proof: in principle this always induces a false proof obligation that the Prover does not demonstrate.
- Or one of the expressions added in the course of the interactive proof is meaningless. Warning: in this case, the proof is not valid (it is nonetheless seldom successful).

The type check of B components discards most of the possibly ill-formed expressions. The remaining problems pertain to the proof and are:

- $\mathsf{card}(E)$ when E is not a finite set.
- the $\mathsf{max}(E)$, $\mathsf{min}(E)$ expressions when E is empty or when E has no maximum nor minimum.
- the $f(x)$ expressions when x does not belong to the f domain, or when f is a relation but not a function.
- divisions by a potentially null expression.
- $\mathsf{size}(s)$, $tail(s)$, etc. when s is not a sequence, such as $\mathsf{size}(\{2 \mapsto 3\})$.

Chapter 5

Tuning-up phase

The key concepts introduced in this chapter are:

Global method: after a re-reading and proof in force 0 of the component, review and examine all POs.

To view a PO, use the PO viewer or the Interactive Prover.
With the Prover: select PO, enter `dd`, use search functions.

Reviewing obligations must first lead to the difficult POs.
The list of obligations is parsed backwards, from the end to the beginning.
Obligations can be reviewed in several phases: quick, final or quick, simplification, final.

A proof obligation is examined in five steps: reading the goal, justification, selecting key hypotheses, intuitive demonstration, notes and tests.
When examining an obligation, the B component must be available.
Reading the goal: it must be interpreted; the verified constraint must be isolated.
Justification: use the component's physical meaning.
Selecting the hypotheses: search at the other end and use the Prover search functions.
Intuitive demonstration: start again with the justification and check used rules.
Notes and tests: note down tentative simplifications and attempt a quick demonstration.

Simplifying a component expression can facilitate its proof.
Every true project is not necessarily demonstrable: under clumsy approaches, it can produce over-complicated proofs.
Divide a project again to simplify its proof.
Write the expressions in the Prover normalized form.
Try and display literal qualities.
Try and write arithmetical expressions under their canonical form.

The quick proof of a PO must be attempted by setting a time-frame.
First try the Predicate Prover.
Do not try and prove more than 5 commands at one shot.
Attempt to generalize a quick demonstration to other obligations.

Obligations with complex expressions can be read by using the Prover as a simplifying tool.
Existential goals are processed using *SuggestforExists*, at times they mark an excess of uncertainty within the component.
Abstract existential goals express the inaccurate expression of an abstract constant.
Undivided goals are often caused by disjunctions.

When a false proof obligation is found, it must be corrected before proceeding.
You must check that the obligation is indeed false as it is not always the case.

Looking for a reverse example is a good means to make sure that an obligation is false. Transferring the reverse example to the component enables the error to be identified.

5.1 Tuning-up global method

The global method to tune-up a component using the proof consists in parsing all proof obligations to check that they are all true. Each time a false proof obligation is met, the component is modified. In order to limit the number of lemmas to read, tuning-up must occur after using the Prover in force 0. The Prover force 0 was devised for such a task, contrary to higher forces that are more time consuming. Force 0 is the best compromise between performance and time to ensure an *a priori* correction of the components (refer to section 4.2).

Tuning-up phase with the Prover can start only **if there are no more visible corrections when simply reading the component**. In other words, the component must have been read again before the tuning-up phase with the Prover. Indeed it is not necessary to use the high level check - the proof - when errors can be detected by a simple reading!

To sum up: the global tuning-up method is as follows:

After reading the component again and **after applying the automatic proof in force 0** ;

- **Parse** the remaining proof obligations (refer to section 5.3) ;
- **Examine** each proof obligation by using the method provided in section 5.4.

These two steps will be studied in detail after this chapter. After, we will review the practical methods to view proof obligations using Atelier B.

5.2 Proof obligations: viewing methods

Two methods are available to review undemonstrated proof obligation with the Prover in force 0:

- **With the PO Viewer**: use Atelier B *PO Viewer* (accessed through the `Status... Show/Print PO` menu in the main window). The*PO Viewer* is only a proof obligation viewer and performs no processing.
- **With the Prover**: enter the Interactive Prover and, for each PO to be viewed, enter `dd` (*Deduction*, to load the local hypotheses and `rp` (*ReducedPo*, a viewing mode reduced to hypotheses having a symbol in common with the goal.

The *PO Viewer* has a faster access than the Interactive Prover, as it has no rule database nor proof strategy, it is only a viewing module and thus much lighter than the Prover. **Viewing with the Prover is required in the following cases**:

- **Complex hypotheses**: if the component structure is such that there are more than about 200 hypotheses and that interesting hypotheses for each proof obligation might not been included in these. In this case, the Interactive Prover search functions will be useful to select hypotheses.
- **numerous proof obligations for one operation**: if there are more than 100 POs for the same operation, using the *PO Viewer* might be difficult as it views proof obligations for an operation in one shot. There is therefore too much information displayed.

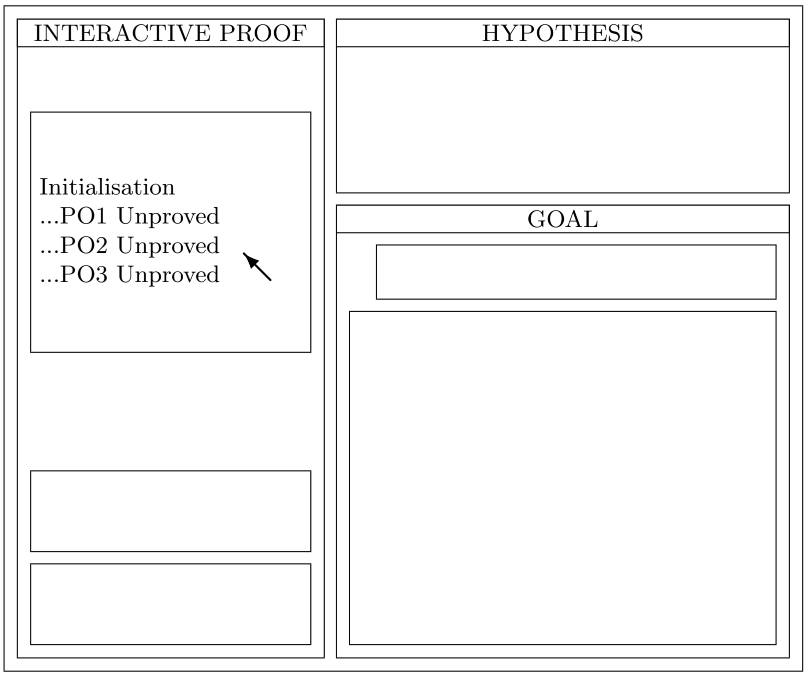
5.2.1 Viewing with the Prover

This is the method to be used when viewing proof obligations with the Interactive Prover:

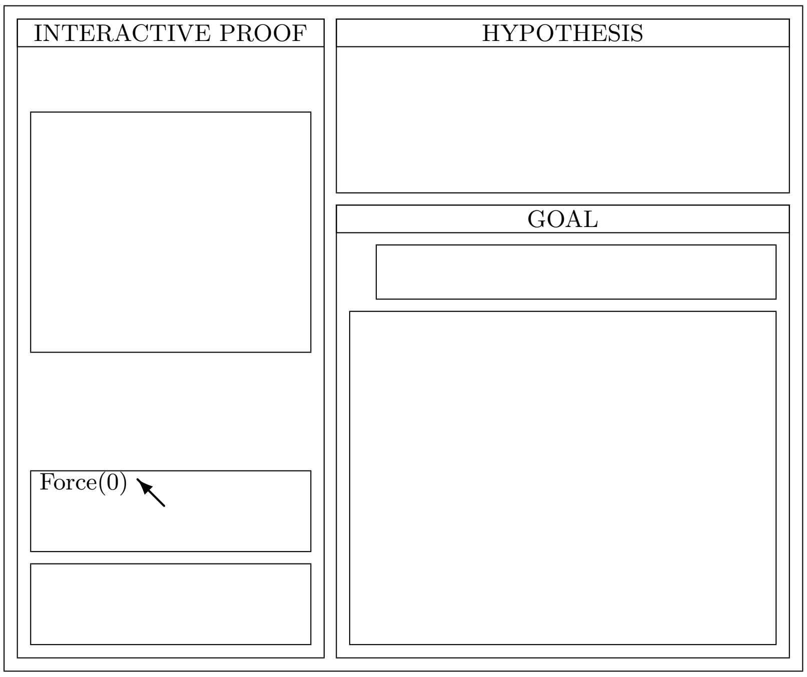
1. Access the selected obligations
2. Load the local hypotheses
3. Use the Interactive Prover search functions

To explain this section, we will perform these three steps using diagrams showing a simplified view of the proof screen. On these diagrams, only the concerned parts will be displayed.

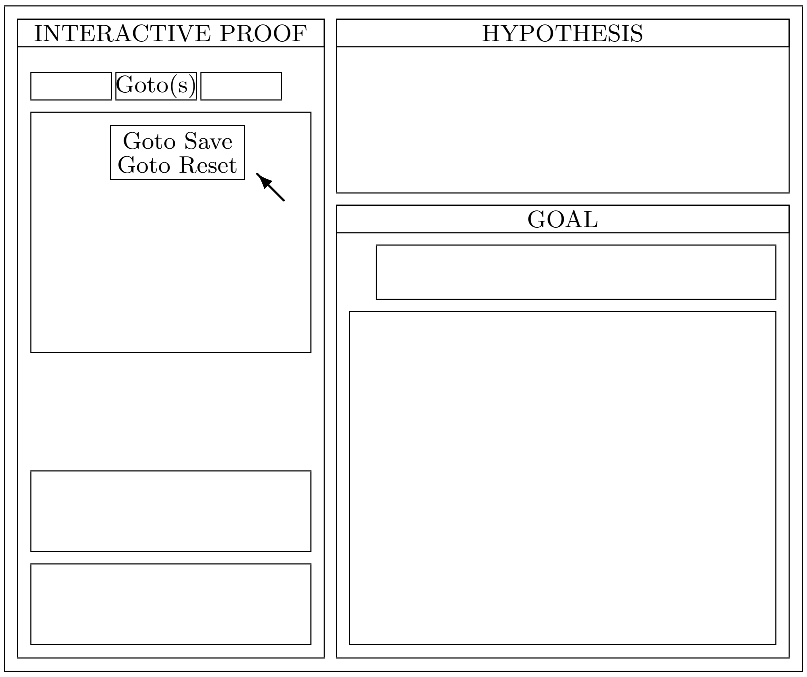
1. **Access the selected obligation**: select the proof obligation to read (this selection is described in the following section) in the list in the global situation window, and double-click it.



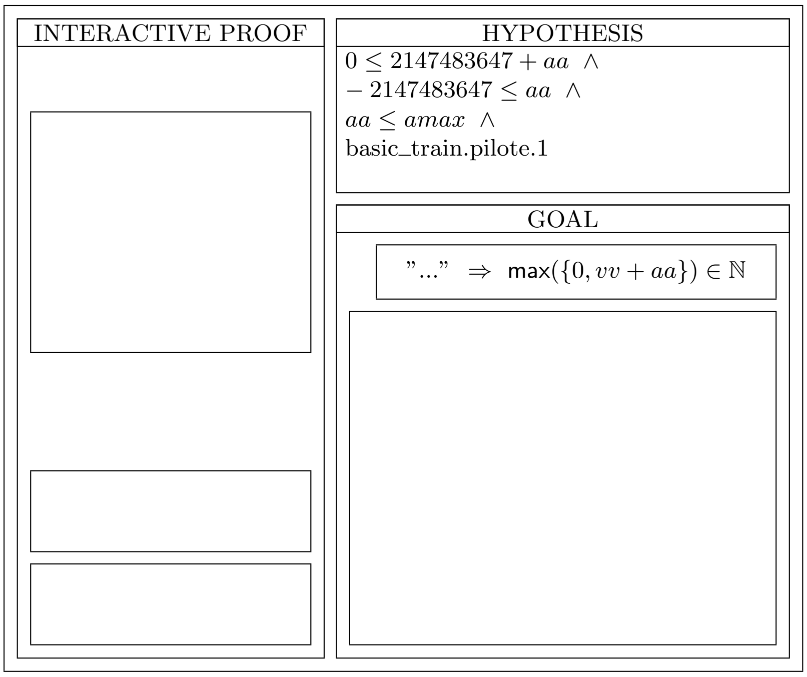
Let us remind you that force has a major influence when loading an obligation (100 hypotheses are loaded in 1 or 2 seconds in force 0 and in 1 minute in force 1!). The recommended force to read a PO is 0 as access time is always negligible and hypotheses are greatly simplified indeed (particularly variables equal between themselves that have been deleted). But we can use the fast force to view unprocessed hypotheses. The saved force on each PO for the interactive demonstration is indeed the one selected in the last interactive session for this obligation. When accessed for the first time, this is force 0. Once the PO is loaded, the force is displayed in the command line area:



When you remember having saved a high force for the obligation to be accessed and you do not want a too long access time, you can use the *GotowithReset* that resets force to 0:



The proof obligation is now loaded; the goal, hypotheses and command areas are filled. The goal is displayed with the local hypotheses for logical proof reasons. Here is a proof obligation display sample, where the sole local hypothesis is a comment between brackets:


INTERACTIVE PROOF

HYPOTHESIS

$0 \leq 2147483647 + aa \ \wedge$
$-\ 2147483647 \leq aa \ \wedge$
$aa \leq amax \ \wedge$
basic_train.pilote.1

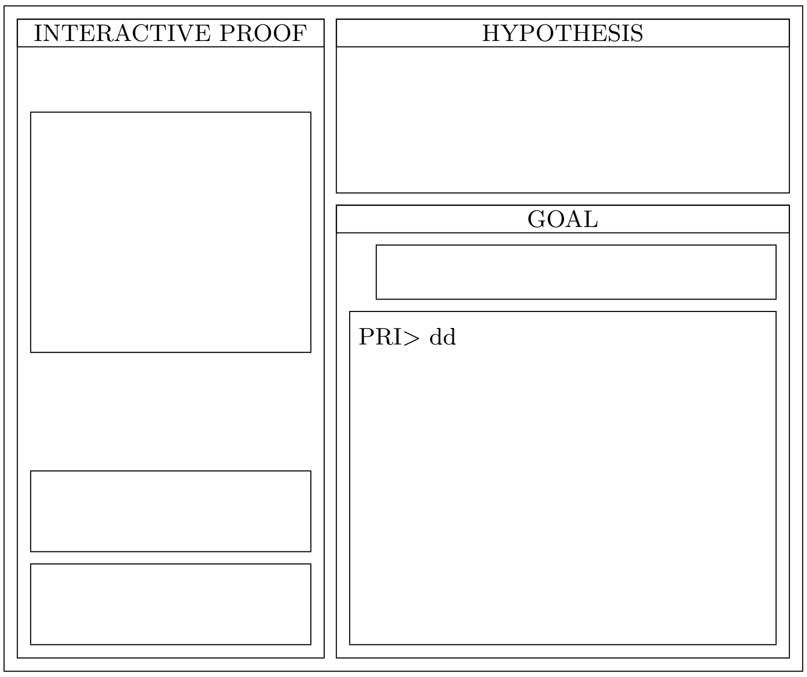
GOAL

$"..." \ \Rightarrow \ \mathsf{max}(\{0, vv + aa\}) \in \mathbb{N}$

Note the localization hypothesis “basic_train.pilote.1”. So we are on the *basic_train* component *pilote* operation and at its first proof obligation.

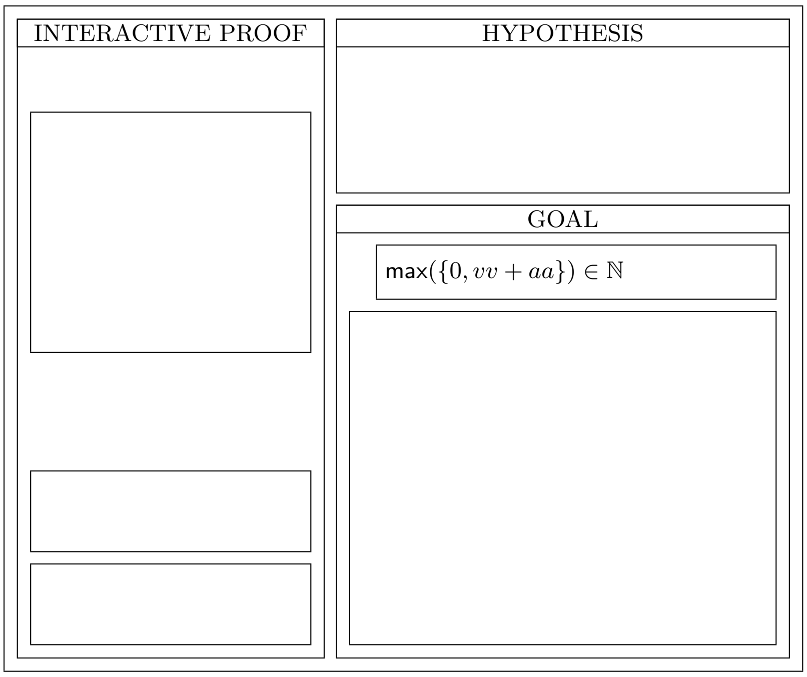
2. **Raise the local hypotheses**: in the Interactive Prover layout, the local hypotheses are displayed with the goal. They must be raised with `dd` (*Deduction*) to ease reading the goal as these hypotheses are only used for the formal proof. Warning: do not save this command when quitting the proof obligation. The Prover will ask the question when you quit this obligation: simply answer no.

   To load local hypotheses, simply enter `dd` in the command window:



```
PRI> dd
```

   The goal is now singled out:



$\mathsf{max}(\{0, vv + aa\}) \in \mathbb{N}$

3. **Using the Interactive Prover search functions**: Remember specifically:

   - the `rp` (*ReducedPO*) function enabling to view only those hypotheses that have a common variable with the goal. In the previous example, we had to demonstrate that $\mathsf{max}(\{0, vv + aa\})$ is a natural, which is always true: but it is nonetheless better to check aa and vv definitions. By entering `rp` in the command area, we get:

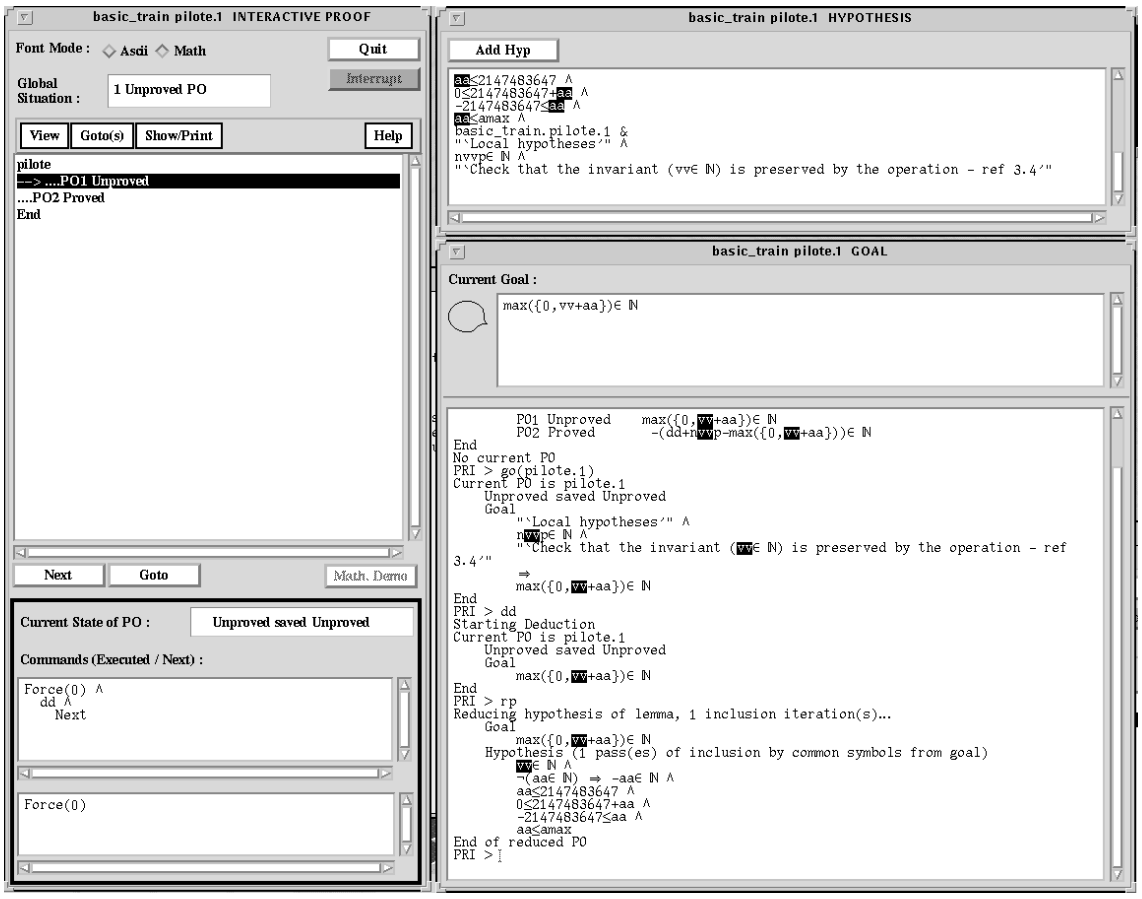
```
PRI > rp
Reducing hypothesis of lemma, 1 inclusion iteration(s)...
  Goal
      max({0, vv + aa}) ∈ ℕ
  Hypothesis (1 pass(es) of inclusion by common symbols from goal)
      vv ∈ ℕ  ∧
      ¬(aa ∈ ℕ)  ⇒  −aa ∈ ℕ  ∧
      aa ≤ 2147483647  ∧
      0 ≤ 2147483647 + aa  ∧
       − 2147483647 ≤ aa  ∧
      aa ≤ amax
  End of reduced PO
PRI >
```

     This time the first two hypotheses define vv as a natural integer and aa as a relative integer (this last form being a Prover normalization: refer to the Prover Reference Manual [?]). The lemma is therefore true.

   - the `sh` (*SearchHypothesis*) function that enables to search hypotheses. You can use this command:
     - in simple mode: for example, `sh(card(EE))` returns all hypotheses that include $\mathsf{card}(EE)$. In our example, we could search for the hypotheses that concern vv: `sh(vv)`, then those that concern aa: `sh(aa)`.
     - in multiple mode: for example `sh(aa _and vv)` for the hypotheses containing both `aa` and `vv`;
     - with patterns: for example `sh(a+b)` for hypotheses that contain additions. One-letter variables are *jokers* and stand for any formula.

   - to select the hypotheses windows and the Interactive Prover command: in these windows press your keyboard CAP key once and the third (center) button on your mouse. Then select *Find All* in the displayed menu. A dialog box is displayed where you can enter a character string for a reverse video display of all its occurrences. This is very useful to rapidly detect all occurrences of an expression in the hypotheses.

As an illustration, you will find the proof area corresponding to our example: a maximum between 0 and $vv + aa$. We have additionaly used the selection function previously described in the hypotheses window to search for aa and in the command window for vv.


basic_train pilote.1 INTERACTIVE PROOF
Font Mode : Ascii Math
Quit
Interrupt
View
Goto(s)
Show/Print
Help
Next
Goto
Math. Demo
basic_train pilote.1 HYPOTHESIS
Add Hyp
basic_train pilote.1 GOAL

Global Situation : 1 Unproved PO

```
pilote
--> ....PO1 Unproved
....PO2 Proved
End
```

Current State of PO : Unproved saved Unproved

Commands (Executed / Next) :

```
Force(0) ∧
  dd ∧
    Next
```

```
Force(0)
```

```
aa≤2147483647 ∧
0≤2147483647+aa ∧
-2147483647≤aa ∧
aa≤amax ∧
basic_train.pilote.1 &
"`Local hypotheses'" ∧
nvvp∈ ℕ ∧
"`Check that the invariant (vv∈ ℕ) is preserved by the operation - ref 3.4'"
```

Current Goal :

```
max({0,vv+aa})∈ ℕ
```

```
        PO1 Unproved    max({0,vv+aa})∈ ℕ
        PO2 Proved       -(dd+nvvp-max({0,vv+aa}))∈ ℕ
End
No current PO
PRI > go(pilote.1)
Current PO is pilote.1
    Unproved saved Unproved
    Goal
        "`Local hypotheses'" ∧
        nvvp∈ ℕ ∧
        "`Check that the invariant (vv∈ ℕ) is preserved by the operation - ref
3.4'"
        ⇒
        max({0,vv+aa})∈ ℕ
End
PRI > dd
Starting Deduction
Current PO is pilote.1
    Unproved saved Unproved
    Goal
        max({0,vv+aa})∈ ℕ
End
PRI > rp
Reducing hypothesis of lemma, 1 inclusion iteration(s)...
    Goal
        max({0,vv+aa})∈ ℕ
    Hypothesis (1 pass(es) of inclusion by common symbols from goal)
        vv∈ ℕ ∧
        ¬(aa∈ ℕ) ⇒ -aa∈ ℕ ∧
        aa≤2147483647 ∧
        0≤2147483647+aa ∧
        -2147483647≤aa ∧
        aa≤amax
End of reduced PO
PRI >
```

5.3 Covering proof obligations

An appropriate selection of the proof obligation reading sequence is useful to target those most likely to be false. An early correction prevents from checking again the *a priori* true proof obligations after the component modification. A bad proof obligation coverage selection leads to a late discovery of errors thus obliging you to read the same obligations several times.

Here are key hints:

1. **Start with proof obligations that seem to be difficult**: These are most likely to be false. This approach is psychologically difficult as the user often wants to eliminate a large number of simple PO "to proceed on"; but this method is advisable. To discover these difficult obligations:

   - **Parse an operation proof obligations from the bottom up**, from the highest number to the smallest. Indeed, the proof obligations generator usually creates most complex proof obligations at the end.
   - **Search the proof obligation concerning the complex parts of the operation.** Using the component structure we can guess what will cause problem.

The order the proof obligations are displayed in the list which depends on the component form. Nonetheless, the proof obligations concerning the operations preconditions called in the operation usually appear at the beginning (even when these calls are at the end of the operation). On the other hand, in an implementation, proof obligations for intermediate computations non-overflow are at the beginning (these obligations are used to check that no overflowing has occurred - for example for xx := 2×v1-v2 we must prove that 2×v1 ∈ INT and 2×v1-v2 ∈ INT).

2. **Do several phases**. It is not necessarily desirable to perform each obligation full analysis immediately by checking these one by one. It often is better to perform several tuning-up phases. Specially when writing components, we perform a reduced tuning-up of goal interpretation completed later.

   - development phase: tuning-up reduced to the sole goals interpretation;
   - final phase: full tuning-up.

   Another method would consist in concentrating on a search for the best form to be given to the component expressions in order to facilitate proof - leading to a separate phase. Indeed, rewriting components to facilitate proof can greatly reduce the number of obligations not demonstrated in force 0 - thus reducing the global cost of proof. For example, we know of a case where writing $e_1 \in \{a, b\}$ instead of $\{a, b\} - \{e_2\} = \{e_1\}$ when a and b are elements of $\{e_1, e_2\}$ brought down the number of non-demonstrated proofs in force 0 from 20 to 0. The components rewriting cycle is:

   - development phase: reduced tuning-up of the sole goal interpretation;
   - simplification phase: tuning-up reduced to a search for the best form to give to the component expressions in order to facilitate proof, without concentrating on the proof obligations correctness;
   - final phase: full tuning-up.

Goal interpretation is one of the five steps when analyzing a proof obligation. We are going to study this now.

5.4 Analyzing a proof obligation

Warning: Before proceeding with a proof obligation analysis, make sure that you have selected this obligation in such a way as to start with those are most likely to detect errors, as explained in the previous section.

In tuning-up phase, you have to get each proof obligation intuitive demonstration as soon as possible. We shall first discuss the principles to analyze a proof obligation, then learn how to parse and view these proof obligations. The method to examine a proof obligation is as follows:

1. **goal interpretation**: examine the various variables present in the goal and find the meaning of each of these in its physical interpretation (refer to section 5.4.1);

2. **intuitive justification**: determine for which reasons this goal must be true within the component context (refer to section 5.4.2);
3. **selecting hypotheses**: in the proof obligation isolate the hypotheses corresponding to these reasons (refer to section 5.4.3);
4. **intuitive demonstration**: perform an intuitive demonstration of the proof obligation reduced to these hypotheses (refer to section 5.4.4).
5. **notes and tests**: the intuitive demonstration can give ideas for the formal demonstration, on the causes of the automatic demonstration failure, etc. In this last step, we want to gain from these ideas. A quick formal demonstration will be looked for and generalized to other obligations - thus enabling to reduce the number of obligations to be examined (refer to section 5.4.5).

This list describes the recommended method for the tuning-up phase. We advise the user to follow these steps for each proof obligation by checking the appropriate sections.

The key idea in this five step method consists in interpreting the proof obligation within the context of the component to be proved. We then benefit from the full intellectual process already performed to understand or build the component, a process that will in time become a set of rigorous demonstrations.

This method requires to read the component to be proven as well as its associated components. We therefore advise you to **save in iconized window the appropriate component file and its associated components**. For an abstract machine, these are seen or included; for an implementation, the superior refinement and the imported machines [1]. Since the Automatic Prover was able to unload proof obligations associated with full parts of the component, **it is sufficient to read a limited part of the component**: the user will get his/her bearings thanks to the operation name (stated in the prover windows label bar) and to the goal.

In the following sections, we will rapidly review a few of the interactive prover commands without making a general presentation of this tool, as for the tuning-up phase it is not necessarily required to know all interactive commands fully. These are studied in detail in chapter 6 which deals with the formal proof.

The five step analysis of a proof obligation might appear lengthy when dealt with in detail - as done here. In fact, with force of habit, each obligation is quickly performed, in a couple of minutes, without necessarily formally separating each step. These steps are only used to tackle problems *in the right order*. If a proof obligation seems to be false then you most probably have met an error in the source: follow the advice given in section 5.8 to find and correct the error.

[1] Notice that even when the interactive prover is started, it is still possible to open the components from Atelier B: simply open the main (iconized) window and double-click on the component.

5.4.1 Hints for the interpretation of goals

The general method to interpret a goal is **to isolate the component concerned part** - usually only one line - and to look for **the constraint to be checked**. It is usually sufficient to concentrate on these two elements to understand a goal origin.

Sometimes the goal takes several lines and a direct reading is not advised. The Interactive Prover functions are then to be used as explained in section 5.7.

When reading a goal, a number of elements must be known for an easy understanding:

- interpreting variables with $: refer to the proof obligations interpretation Manual. Without duplicating this document, here are a few simple rough landmarks:
  - vv variables: when proving a refinement or an implementation, most abstract level variables.
  - vv$0 variables: in the case of a variable modified several times (loop or sequence), initial value.
  - vv$1 variables: refinement or implementation variables that are being proved or variables from an imported machine.
  - vv$2 variables: variables within a loop.
  - vv$7777 variables: variables at the end of the operation modifications (e. g. after a loop).
- imported operations: to generate an implementation proof obligations, the PO generator textually expands the imported operations. We therefore have to expect and find in these obligations the imported operations code. For example, when an operation specified by `ANY xx WHERE ...` is used, the `xx` intermediate variable can appear in the user implementation proof obligations. The previous remark is true at all levels for included operations.
- positioning in cases: when specification or refinement contains cases, the PO generator separates the different cases into different proof obligations. Thus it is necessary to look into the local hypotheses as these specify in which case we are, and allow us to find the correct line. Here are examples that induce separations by cases:
  - ANY ... WHERE $P_1 \Rightarrow Q_1 \wedge P_2 \Rightarrow Q_2$ ... in a specification.
  - SELECT P_1 THEN S_1 WHEN P_2 THEN S_2 END in a specification.
  - IF ... THEN ... ELSE .. END in an implementation.
  - CASE in an implementation.
- eliminating variables: the proof obligations generator deletes all unecessary intermediate variables. For example, for VAR *vv* IN *vv* $\longleftarrow$ *op* when the specification of *op* is *rr* $\longleftarrow$ *op* = *rr:=var*, it is *rr* and not *vv* that appears in the proof obligations. This might seem surprising as it is not the data handled in the component that appears.
- order of proof obligations: Proof obligations concerning the pre-conditions of operations called by the clause to be proven are usually displayed at the beginning (even

when these calls are at the end of the clause). In an implementation, intermediate computations non-overflow proof obligations are situated at the beginning.

- commenting the goal: the hypothesis displayed at last in the hypotheses window after you have entered `dd` is a comment explaining goal origin. This comment gives an explanation and a reference to the proof obligations Manual.

Example: we assume that proof obligation goal 30 of the *Commandes Pompes* operation of the *CmdPmp_1* component, that is:

$(1\,..\,NB\_PUMP) \times \{FALSE\} \lhd\!\!-$
$(indice\_l\$7777 + 1\,..\,NB\_PUMP \lhd \{\mathsf{TRUE} \mapsto (l\_wpok \rhd \{\mathsf{TRUE}\}), \mathsf{FALSE} \mapsto \varnothing\}(on\$1)) =$
$(1\,..\,NB\_PUMP) \times \{FALSE\} \lhd\!\!- \{\mathsf{TRUE} \mapsto (l\_wpok \rhd \{\mathsf{TRUE}\}), \mathsf{FALSE} \mapsto \varnothing\}(on\$1)$

The four variables used in this goal are NB_PUMP, l_wpok, $indice\_l\$7777$ and $on\$1$. First we find the meaning of each variable: NB_PUMP is the number of pumps to control, l_wpok is a table that indicates for each pump if it is OK or not, $indice\_l\$7777$ is a course index for each pump and $on\$1$ represents a meter status. Examine the component; the operation contains a loop whose index is $indice\_l$. The presence of \$7777 shows that this obligation concerns the status after leaving the loop. Let's check that this loop has duly built what was anticipated in the specification. The goal comment is:

Check that the invariant (pumpon = pumpon\$1) is preserved by the operation

In this operation specification we had indeed written:

$pumpon := (1\,..\,NB\_PUMP) \times \{FALSE\} \lhd\!\!- \{\mathsf{TRUE} \mapsto (l\_wpok \rhd \{\mathsf{TRUE}\}), \mathsf{FALSE} \mapsto \varnothing\}(on\$1)$

This shows that this variable must be built from the table where all pumps are set to FALSE adding that if $on = \mathsf{TRUE}$ the table l_wpok. Goal interpretation is ended.

A correct goal interpretation requires knowledge of the component and its adjacent levels. We recommend keeping these files opened in icons in a corner of the screen during proof phases (Caution: do not leave these displayed: proof windows must always stay visible and side by side. It is often better to read these components again when begining a proof session. The meaning of many of these Pos is then clear without requiring to look at the files again.

Notice that once the interactive prover has been launched, it is still possible to open components from Atelier B: simply open the iconized main window and click twice on the component.

Once the goal meaning is understood, we can inquire why it must be true: this is the intuitive justification.

5.4.2 Hints for the intuitive justification

- **Use the physical meaning**: think about what the quantities and expressions being handled represent physically: for example $vv + (aa * tt)$ corresponds to a new speed,

$RESSOURCES - \{x_0\}$ is the new set of resources, ... At this stage, mathematical proof is not sought, but a sound interpretation of the abstract model.

- **Reason by cases and by contradiction**: we very often overlook these two tricks in natural reasonings. They usually apply well when direct justification is not successful.
- **Take note of the intuitive justification**: It will be used for the intuitive demonstration (Make a rough note).

Example: let us resume with our previous example. Why have we built at the end of the loop the value of *pumpon* anticipated in the specification ? Simply because each pump was checked (using the physical meaning). This means that *indice_l*\$7777 - representing the loop index at the end of the loop - must be such that all pumps are checked. By looking at the operation, we see that this index goes through pump numbers from top to bottom; it is nil at the end of the loop. Thus the $\$7777 + 1 \,..\, NB\_PUMP$ interval is equal to $1 \,..\, NB\_PUMP$, this interval is used to restrict the right part's expression that is evidently a function of $1 \,..\, NB\_PUMP$ in BOOL. This restriction can thus be removed and the goal becomes a literal equality. Notice: replacing the index with its value deletes the restriction in the left part. We did not need a reasoning by contradiction here.

The proof obligation intuitive justification phase can be the most important: this is when we perceive all consequences of the selected B modeling. We now understand why the proof obligation must be true within the component context. This context must also be found in the hypotheses that we are going to examine.

5.4.3 Hints for the selection of hypotheses

The key point is as follows:

- **read the hypotheses from the bottom**. In the goal the most significant hypotheses are usually the ten last ones. The context hypotheses begining the list are usually constant properties with no relation with the goal!

On the other hand, the hints on reading the goal apply also when reading the hypotheses. Since the hypotheses list is often very lengthy, we must **know what we are looking for** before any reading; it is by the intuitive justification of the proof obligation that the search for significant hypotheses is conducted. When the search is led within the Interactive Prover, use the search functions described in section 5.2.1 as much as possible.

When the significant hypotheses are identified, you have to **note** the information required to find these again. This will be useful during the intuitive demonstration phase.

Example: let us resume with the previous example. We are looking for:

- the hypothesis indicating that the index is nil ;
- the hypothesis indicting *l_wpok* type.

We can try the *ReducePo* (`rp`) command. If only a few hypotheses are selected, we would have found those we are looking for as they share at least the name of the concerned variable with the goal. Indeed the `rp` command returns:

$$\boxed{\begin{array}{l} NB\_PUMP = 4 \wedge \\ l\_wpok \in 1 \mathinner{\ldotp\ldotp} 4 \;\mathrel{+\!\!\!\!\to}\; \mathsf{BOOL} \wedge \\ \mathsf{dom}(l\_wpok) = 1 \mathinner{\ldotp\ldotp} 4 \wedge \\ on\$1 \in \mathsf{BOOL} \wedge \\ on\$1 = \mathsf{TRUE} \;\Rightarrow\; off\$1 = \mathsf{FALSE} \wedge \\ on = on\$1 \wedge \\ indice\_l\$7777 = 0 \wedge \\ indice\_l\$7777 \in 0 \mathinner{\ldotp\ldotp} NB\_PUMP \end{array}}$$

We have found all the useful hypotheses. Notice that useful hypotheses are found with a simple `rp` command.

5.4.4 Hints for the intuitive demonstration

Intuitive demonstrations are aimed at avoiding the wrong belief that some proof obligations are true - following confusions easily made in an intuitive justification. The general method is as follows:

- **transposing the intuitive justification** by using the selected hypotheses
- **invoke rules being used**: we do not want to return to rules present in the rules library or in the B-Book, but only check which rules are actually used in the intuitive obligation.
- **examine these rules**: it is when the transformation was used outside of its context as a general rule, that we can see all the conditions required for its validity.

Example: we sum up our previous example. The justification steps were:

- replace index with its value: the rule used is the standard definition of equality.
- simplify $0 + 1 \mathinner{\ldotp\ldotp} NB\_PUMP$ into $1 \mathinner{\ldotp\ldotp} NB\_PUMP$: integer calculus rules.
- eliminate the restriction to $1 \mathinner{\ldotp\ldotp} NB\_PUMP$: we have an expression of the form

  $$1 \mathinner{\ldotp\ldotp} NB\_PUMP \lhd f$$

  f being defined by

  $$f = \{\mathsf{TRUE} \mapsto (l\_wpok \rhd \{\mathsf{TRUE}\}), \mathsf{FALSE} \mapsto \varnothing\}(on\$1)$$

  the rule to apply is clearly:

  $$f \in A \;\mathrel{+\!\!\!\!\to}\; B \;\Rightarrow\; A \lhd f = f$$

  but how do we prove that f is a function beginning at $1 \mathinner{\ldotp\ldotp} NB\_PUMP$? We have to create two cases: $on\$1 = \mathsf{TRUE}$ or FALSE.

  - if $on\$1 = \mathsf{FALSE}$: f is empty, the restriction is immediately deleted.
  - if $on\$1 = \mathsf{TRUE}$: $f = l\_wpok \rhd \{\mathsf{TRUE}\}$. Reasoning is a little more complex, we will have to use the typing rules of a co-restricted function. In an intuitive demonstration, it is not necessary to develop everything, it is enough to display the key rules used.

- once the restriction is deleted, we have two literaly equal terms (rule used: the definition of equality).

During this process, beware of the standard traps given hereafter.

Standard traps

- **priority $\Rightarrow$ and** $\wedge$: implication has less priority than conjunction. For example:
  $A \ \wedge$
  $B \ \Rightarrow \ C \ \wedge$
  D
  will be understood as $(A \ \wedge \ B) \ \Rightarrow \ (C \ \wedge \ D)$. To avoid a bad interpretation, place between parenthesis. Write, for example:
  $A \ \wedge$
  $(B \ \Rightarrow \ C) \ \wedge$
  D

- **empty intervals**: an interval can be empty when its left bound overcomes its right one. For example: $aa \ \in \ aa..bb$ is not always true.

- **adding divergences to a function**: if *ff* is a function, the object produced by adding ordered pairs to *ff* might no longer be a function as one of the added ordered pair can have the same beginning element than an ordered pair existing in *ff*. In this case, *ff* is transformed into a relation.

- **integer division**: in B, the / symbol denotes the integer division. Calculus rules are much less reduced than with real division. For example, we do not have $(a + b)/c \ = \ a/c \ + \ b/c$ (reverse example $a = b = 1$, $c = 2$). Or also, we do not have $0 \leq a/2 \ \Rightarrow \ 0 \leq a$ (reverse example with $a = -1$).

- **confusion of levels in functions of functions**: functions of functions often produce more complex predicates than expected. For example, we have
  $f_0 \ \in \ \mathrm{INDEX} \ \ \rightarrow \ \ (\mathrm{NAT} \nrightarrow \mathrm{NAT})$
  that we want to refine by
  $f_1 \ \in \ \mathrm{INDEX} \rightarrow \ (\mathrm{NAT} \ \ \rightarrow \mathrm{NAT})$
  The link invariant is not $f_0 = \mathsf{dom}(f_0) \lhd f_1$, but

$$\forall x.(x \in \mathrm{INDEX} \ \Rightarrow \ f_0(x) = \mathsf{dom}(f_0(x)) \lhd f_1(x))$$

- **extensions with variables**: When the elements of an extended set are not literal (algebraic) expressions, these elements may be evenly matched. Forgetting this leads to errors: for example $\{xx, yy\} - \{zz\}$ is not equal to $\{xx, yy\}$ (notably when $zz = yy = xx$ is the empty set).

5.4.5 Notes and tests

During the four previous steps, the user devised simplifications for the expressions he/she examines and beginnings of interactive demonstrations. Before leaving a proof obligation, it is thus advised to:

- **Reconsider the expressions form**, specifically, start the Prover in force 0 once, to find out which is the first failed goal. This helps to find simple expressions helping the Prover task. Follow the hints given in section 5.5.
- **Try a rapid demonstration**. When successful, the obligation is discarded and its demonstration can be generalized, deleting thus other obligations. Warning: do not in any case start a formal proof phase! Follow the advice given in section 5.6.

About the example we have been studying - for each step we can make remarks on the form of the expression appearing in the goal:

$$(1 \mathinner{..} \mathit{NB\_PUMP}) \times \{\mathit{FALSE}\} \lhd \{\mathsf{TRUE} \mapsto (\mathit{l\_wpok} \rhd \{\mathsf{TRUE}\}), \mathsf{FALSE} \mapsto \varnothing\}(\mathit{on}\$1)$$

It might be more natural to explicitly display the two cases according to $\mathit{on}\$1$. We would thus get two simpler expressions:

$$(1 \mathinner{..} \mathit{NB\_PUMP}) \times \{\mathit{FALSE}\} \lhd (\mathit{l\_wpok} \rhd \{\mathsf{TRUE}\})$$

and

$$(1 \mathinner{..} \mathit{NB\_PUMP}) \times \{\mathit{FALSE}\}$$

A first expression is in fact simplified since $\mathit{l\_wpok}$ is a total function of $1 \mathinner{..} \mathit{NB\_PUMP}$ in BOOL. It simply becomes $\mathit{l\_wpok}$!

Do we have to make a note of these simplifications? This is not obvious. But if we decide to perform an expression simplification phase, these ideas would certainly be useful, they should then be noted down.

We can also try a rapid demonstration of this obligation. As we examine this PO we are induced to make two cases according to $\mathit{on}\$1$ value. In fact, by starting this proof by cases using the *DoCases* command, we directly succeed. When used on the interactive demonstration, it is possible to have a hint of it. Unfortunately, it is not generalized to the component undemonstrated obligations.

5.4.6 Admitting proof obligations

It is possible to use an admission rule (see this concept in 6.6.2) to admit proof obligations. This method consists in having a manual rule artificially demonstrate the proof obligations that cannot be seen. It has two advantages:

- proof obligations thus discarded are duly ticked as proven - for example, the *Next* command cannot reach them.

- the Prover higher forces could be launched on non-discarded proof obligations without loosing time on the admitted ones.

But, application of the admission rule is saved as a manual demonstration for the concerned proof obligations - which can be a problem when the component is modified. Indeed, the proof obligations can then have been modified and application of the admission rule can lead to think they are true. Using an admission rule requires a fair knowledge of the Interactive Prover. Here, we shall only see when this method applies during a tuning-up phase. To learn how to use admission rules, read section 6.

5.5 Simplifying B component expressions

Some components create too many proof obligations or too complex ones. It even happens that the number of proof obligations to be generated for a component is so large that the generator saturates and fails. In all these cases it is on the components themselves that we must act to select the forms of the expressions facilitating the proof.

On the other hand, simplifying the expressions of formal specifications often enables to understand the problem better, as the contemplated software is then better modeled, in a form *that facilitates reasoning* as it facilitates the proof. The spirit of B method is to build the software in order to prove it satisfies the considered need (read the B-Book foreword).

It must be clear that **not every true project is necessarily demonstrable**. Proof is a high level check that can only be performed when various separation requirements are respected (*Divide and Conquer*). A standard example of this, is the analysis of proof obligations concerning operations in sequence after an IF: these POs are split n times, n being the number of IF pathes.

Transforming the component to return to expressions that facilitate the proof is a complex methodological subject that we will study very briefly here. We will merely give a number of “recipes”.

5.5.1 Separating components

Proof obligations too numerous or too complex often denote a bad separation of the components. Instruction sequences will then be too lengthy (see above), nested loops, etc. A new separation is the first method to try to reduce proof difficulty. Think specially about :

- Create imported machines to separate the implementation complex parts in major operations.
- Insert refinements when a proof implementation is too complex.

5.5.2 Take into account Prover normalization

The Prover performs a number of normalizations in a way sometimes partial for performance reasons. In a tuning-up phase, it could be useful to modify the component expressions to rediscover the Prover normalization. This can greatly influence the performances. The method is as follows:

1. **consult the Prover**: apply the Prover on a non-demonstrated proof obligation, in force 0, to examine the goal and the failed hypotheses and check under which form it has written the component expressions (warning: enter `pr` without previously entering `dd`, else the local hypotheses are raised non-normalized);
2. **normalize**: write the component expressions in the equivalent forms given by the Prover;
3. **test**: restart Prover in force 0 (refer to section 4.2) and note gain or loss.

In addition to this rather experimental process, we are hereafter going to give a number of precautions to follow systematically. The Prover normalization are explained in the "Normalization" chapter, Prover Reference Manual.

5.5.3 Search for literal equalities

The Prover being mechanical, its basic principle is to have formulas coincide. For this reason, we must avoid wantonly modifying the component expression's form. For example, when we have in the invariant:

$$\{a \mapsto b\} \in f \quad \Rightarrow \quad P$$

And when in a pre-condition we want to place the case where $\{a \mapsto b\} \in f$, we certainly must not write:

$$\text{PRE } a \in dom(f \rhd \{b\})$$

But rather:

$$\text{PRE } \{a \mapsto b\} \in f$$

The above example is intentionally exaggerated. This type of problem can nonetheless occur when writing a component - specially when this one has been successively greatly modified. We must also avoid using intermediate constants that would oblige the Prover to perform replacements. Instead we use a clause DEFINITION that performs a literal replacement. For example, do not write:

CONSTANTS card_ens **PROPERTIES** card_ens = card(ens)

But rather:

DEFINITIONS card_ens == card(ens)

5.5.4 Search for arithmetic expressions in canonical form

All Prover forces use the *Arithmetic Solver* that sets the arithmetic expressions in a canonical form - the variables having a constant order between themselves. Success through literal correspondence is thus augmented.

In force 0 and for efficiency reasons, only the fully arithmetic predicates - ($a = b$ or $a \leq b$) - are transformed by the *Arithmetic Solver*. It might be that the raw form is displayed in the proof obligation mingled with the canonical form, creating annoying asymmetries. For example:

$$jj + 1 \leq ii \;\Rightarrow\; ii..jj + 1 = \varnothing$$

will be viewed by the Prover as:

$$1 + jj \leq ii \;\Rightarrow\; ii..jj + 1 = \varnothing$$

The Prover normalization has inverted jj and 1 terms. The simple rule ($i..j = \varnothing \Leftrightarrow j + 1 \leq i$) thus cannot be directly applied.

It is not then always possible to directly write the arithmetic expressions in their canonical form. We shall content ourselves to advise **writing the numerical constants** on the left and **arranging the variables in the components' arrival order**, that is to say, the order in which these variables are displayed in the invariant.

5.6 The quick proof

We denote by *quick proof* all interactive demonstrations discovered in a few minutes during the tuning-up phase. These are very simple demonstrations but they prevent from returning to the specified proof obligation and - thanks to their simplicity - they can be generalized to other obligations. We shall describe this process in two steps: finding a quick demonstration and generalizing it.

5.6.1 Finding a quick obligation

- **Setting a time limit**: the user must avoid getting lost in a true formal proof - he/she will set a time limit - from 30 seconds to 2 minutes according to the obligation complexity.

- **Using the Predicate Prover**: when the obligation appears to be demonstrable without arithmetical concepts, you can try the *PredicateProver* (`pp`) command that starts the Predicate Prover with a 60 seconds time limit. It is usually better to use this command after a (`dd`) deduction to load the local hypotheses. Some useful variations:
  - `pp(30)` to reduce expiration time to 30 seconds ;
  - `pp(rp.1)` or `pp(rp.2)` enables to apply the Predicate Prover on a reduced obligation - with one or two hypotheses inclusion passes from the goal. Indeed, if you have more than a hundred hypotheses, the `pp` command will be lengthy: the sole translation time into quantified predicates is already significant.
- **5 commands is a limit**: when the Predicate Prover has not unloaded the obligation, try a demonstration from the Interactive Prover standard commands. We shall not explain here how to proceed, this is done in chapter 6. We only note that the two most frequently found commands in a quick demonstration are *AddHypothesis* (`ah`) and *DoCases* (`dc`).

If you have never used the Interactive Prover and have not yet studied chapter 6 on formal proof, you will not perform quick demonstrations when tuning-up your first component. In this case, it might be interesting to do a formal proof phase just after tuning-up your first component - as a didactic exercise -even if you had planned a global tuning-up of the whole project. In any case, never start a formal proof before tuning-up at least one component.

5.6.2 Generalizing a demonstration

To generalize a demonstration, use the *TryEverywhere* (`te`) command. We describe it here as it is often the quick demonstrations that are generalized.

- **Save your demonstration** before generalizing with the *SaveWithoutquestion* (`sw`) command. Or you could loose it when generalizing.
- Using *TryEverywhere*:
  - `te(Op.30, Replace.Gen.Unproved)`: try applying operation *Op* obligation 30 demonstration to all unproved component POs. Let us remind that the operation name and the current obligation number are displayed in the three Prover window title bars as well as in all answer messages in the command area.
  - `te((dd & pp), Replace.Gen.Unproved)`: try the demonstration made up of a deduction and start the Predicate Prover on all unproved component POs.

For additional information on the *TryEverywhere* command, refer to the Prover Reference Manual.

5.7 Complex expressions

In some cases the goal is written on several lines and a direct reading is not possible. You must then start its demonstration using the Interactive Prover to split it up and make it understandable. The existence of a complex goal often shows that the form of the component expressions must be modified to simplify the proof. In any case the goal must be analyzed before any modification. Complex goals usually belong to one of the following categories;

- Particularizable existential goals: the goal has this form: $\exists x.P$ - but one can find a convenient x simple value:
- Abstract existential goals: the convenient x value is unique and complex, usually a λ function;
- Non-separated goals: goals that produce a failure of the proof obligation generator separation algorithm.

Now we are going to study these three categories. In the examples that illustrate the following sections, we no longer use the *Deduction* (`dd`) to load local hypotheses but the `pr(Red)` command that starts the Automatic Prover in reduced mode. In interactive proof, we use `pr(Red)` to proceed with an hypothesis without completing it or load hypotheses - for reasons that will be discussed lengthly in the chapter 6.

5.7.1 Particularizable existential goals

Let us explain this category with examples:

Example 1: an obvious value has to be found. We have the following components:

MACHINE ComplexGoal1 **ABSTRACT_CONSTANTS** cc **PROPERTIES** cc ∈ 1..3 ⇸ NAT ∧ ∀xx.(xx ∈ ran(cc) ⇒ xx mod 2 = 0) ∧ ∀xx.(xx ∈ ran(cc) ⇒ xx mod 3 = 0) ∧ ∀xx.(xx ∈ ran(cc) ⇒ xx mod 5 = 0) ∧ ∀xx.(xx ∈ ran(cc) ⇒ xx mod 7 = 0) **END**	**IMPLEMENTATION** ComplexGoal1_1 **REFINES** ComplexGoal1 **END**

Artificially, this component performs no operation but represents the real case where an abstract constant is used to specify the software without being used for the implementation. For the implementation, we have a proof obligation whose goal is:

$\exists cc.(cc \in 1\,..\,3 \ \nrightarrow\ \mathsf{NAT} \ \wedge\ \forall xx.(xx \in \mathsf{ran}(cc) \ \Rightarrow\ xx\mathsf{mod2} = 0) \ \wedge$

$\forall xx.(xx \in \mathsf{ran}(cc) \;\Rightarrow\; xx\mathsf{mod}3 = 0) \;\wedge\; \forall xx.(xx \in \mathsf{ran}(cc) \;\Rightarrow\; xx\mathsf{mod}5 = 0) \;\wedge$
$\forall xx.(xx \in \mathsf{ran}(cc) \;\Rightarrow\; xx\mathsf{mod}7 = 0))$

As we see, we only have to prove our specification validity by showing that there is a constant verifying the expected properties. Here we have only to take $cc = \varnothing$. We use the *SuggestforExists* (`se`) command enabling to suggest attempting the proof with a specific value:

`pr(Red)`	to load local hypotheses
`se({})`	suggesting to take $cc = \varnothing$
`pr`	restarts the Automatic Prover

And the proof directly succeeds. If it was not the case, we should examine all goals produced with the *SuggestforExists* command by possibly using the technique described in section 5.7.3. Such a type of complex goal does not point out the goal being modified.

Example 2 : too unspecified form. We have the following components:

MACHINE ComplexGoal2 **OPERATIONS** xx ⟵ op = ANY ee WHERE ee ⊆ NAT ∧ ∀uu.(uu ∈ ee ⇒ uu mod 3 = 0) THEN xx :∈ ee END **END**	**IMPLEMENTATION** ComplexGoal2_1 **REFINES** ComplexGoal2 **OPERATIONS** xx ⟵ op = xx := 9 **END**

The above specification simply means that the operation must return a result divisible by 3 - but it is expressed under a too much unspecific form. We get the following goal:

$\exists ee.(ee \subseteq \mathsf{NAT} \;\wedge$
$\forall uu.(uu \in ee \;\Rightarrow\; uu\mathsf{mod}3 = 0) \;\wedge\; \exists(xx\$0).(xx\$0 \in ee \;\wedge\; 9 = xx\$0))$

In order to read this goal more easily, we ask the Prover to start the proof in reduced mode: `pr(Red)`. The goal is now:

$\exists ee.(ee \subseteq \mathsf{NAT} \;\wedge\; \forall uu.(uu \in ee \;\Rightarrow\; uu\mathsf{mod}3 = 0) \;\wedge\; 9 \in ee)$

This amounts to prove that 9 is in NAT and that 9 can be divided by 3! Faced with such a wanton type of indetermination, we have to modify the specification. The operation specification is now:

$xx : (xx \in \mathsf{NAT} \;\wedge\; xx\mathsf{mod}3 = 0)$

This time, the set is demonstrated in force 0 without user interaction.

Conclusions: When an existential goal can be solved by an obvious value or a specific value stemming from the specification, we have to check whether the non-specification producing this existential goal is indeed needed. When this is the case, we have to demonstrate the proof obligation by using the *SuggestforExists* command - otherwise, we have to delete the useless non-specification.

5.7.2 Abstract existential goals

These are existential goals generated by mathematical abstract constants. Let us assume that you need the notion of factorial in a specification. You can describe this notion as an abstract constant:

ABSTRACT_CONSTANTS
  fact
PROPERTIES
  $\text{fact} \in \mathbb{N} \rightarrow \mathbb{N} \wedge$
  $\forall nn.(nn \in \mathbb{N} \Rightarrow \text{fact}(nn+1) = (nn+1)\times\text{fact}(nn)) \wedge$
  $\text{fact}(0) = 1$

As this constant will indeed never be implemented, we will have to prove that it exists. The concerned application will appear rather late in the development cycle - when writing the component implementation - as it is only then that Atelier B knows that the constant will not be implemented. The goal will be:

$\exists\text{fact}.(\text{fact}\in \mathbb{N} \rightarrow \mathbb{N} \wedge$
$\forall nn.(nn \in \mathbb{N} \Rightarrow \text{fact}(nn+1) = (nn+1)\times\text{fact}(nn)) \wedge$
$\text{fact}(0) = 1$

Such a demonstration forces us to write the factorial as a direct definition, as the sole means to prove an existential goal is to propose a value. Here:

$\text{fact}= \lambda nn.(nn \in \mathbb{N} \mid \prod(ii).(ii \in 1\,..\,nn \mid ii))$

As we have to write this definition, it is better to have it in the component. It is nonetheless useful to keep the properties previously written as hypotheses for the proof - this replacing the factorial mathematical knowledge absent from the Prover. Indeed we will have to prove these properties. We can write:

ABSTRACT_CONSTANTS
  fact
PROPERTIES
  $\text{fact} \in \mathbb{N} \rightarrow \mathbb{N} \wedge$
  $\forall nn.(nn \in \mathbb{N} \Rightarrow \text{fact}(nn+1) = (nn+1)\times\text{fact}(nn)) \wedge$
  $\text{fact}(0) = 1 \wedge$
  $\text{fact}= \lambda nn.(nn \in \mathbb{N} \mid \prod(ii).(ii \in 1\,..\,nn \mid ii))$

In this case, we shall have to prove the factorial properties when implementing the component, or:

ABSTRACT_CONSTANTS
  fact
PROPERTIES

fact= $\lambda nn.(nn \in \mathbb{N} \mid \prod(ii).(ii \in 1\,..\,nn \mid ii))$
ASSERTIONS
fact $\in \mathbb{N} \rightarrow \mathbb{N} \wedge$
$\forall nn.(nn \in \mathbb{N} \Rightarrow \text{fact}(nn+1) = (nn+1)\times\text{fact}(nn)) \wedge$
$\text{fact}(0) = 1 \wedge$

In this case, we shall perform the demonstration during the component proof, that is to say immediately.

Conclusion: faced with an abstract existential goal, the component will always have to be modified to produce an explicit definition of the concerned mathematical notion.

5.7.3 Non-separated goals

Non-separated goals appear when the separation algorithm in the obligation generator fails. It is not necessarily needed to modify the component - such obligation can be easier to prove that it seems. In most cases, the non-simplified goal has the form: $P \vee Q$ To read such goals, we must use the `pr(Red)` command to load local hypotheses while simplifying the goal.

5.8 Proof obligations appearing to be false

When the component being examined contains errors, some proof obligations are false. This is how the B method indicates errors such as ill-respect of a specification or of invariant properties. We recommend the following method when meeting with false proof obligations:

- **Make sure that the proof obligation is really false**: when a proof obligation seems false, we must be sufficiently certain before modifying any components. Ill-decided modifications can have very costly consequences.
- **Find the mistake in the component**: simply check with the component context the reasons why the proof obligation is false. Notably, when we have a reverse example, report the values in the component.
- **Correct**:a false PO can vary from a mere neglect to the obligation of reconsidering the current mathematical model. This is why the false proof obligations must be detected in a continuous process - it is not wise to imagine that no component modifications are necessary before the proof phase. Corrections after a false PO belong to the B method; we simply give the following advice :
  - check whether if, in the physical meaning given to the model variables, the values making the proof obligation false, are possible. Otherwise, the invariant is too weak.
  - when led to reinforce a too weak invariant, check that the new property is always true for all operations. Indeed, it often happens that a physically true

  property forgotten in the invariant is not kept by operations whose coding is avoided by a too weak invariant.
  - before modifying an invariant, check that the modification makes the false obligation become true ! Do not *a priori* modify to 'see what it does' to the PO's.
  - note the mathematical reasoning used. Generally, justifications received during a component tuning-up ease the subsequent proof phases.

We now study in detail how to avoid wrongly considered proof obligations as false .

5.8.1 Make sure that the proof obligation is false

To avoid dangerous modifications in the component, we have to check that the proof obligation that seems false is indeed false. To do this, consider the following points:

- **Check that the proof obligation is not made true by contradictory hypotheses being present**: it is normal that such POs do appear - a proof of the rigorous application of the theory. These proof obligation goals do not have to be true - they can even be meaningless. In any case, we are in contradiction between a component path and its higher level. It is by following this path that the contradiction will be discovered.
- **Look for a contrary example**: look for specific variables values that verify the hypotheses but not the goal. Pay attention to the following points
  - try and use the 0 et $\varnothing$ values. They often enable to create simple reverse examples.
  - when there are many hypotheses, find a reverse example only for the variables active in the goal. Do not try to check whether the proof obligation is true by contradictory hypothesis - by looking for the reverse example - this is too complex.
- **meaningless proof obligations**: it is certainly possible for a proof obligation to be meaningless. Such obligations do indicate an error in the component but they can be surprising mainly because there is no clear contrary example. Thus:

  $uu \in \mathbb{N} \ \wedge \ uu \leq 10 \ \Rightarrow \ uu\$2 \in \mathbb{N}$

  A reverse example is $uu\$2 = \mathsf{TRUE}$ and$uu = 0$. Looking for such a type of contrary example can be disconcerting. The previous contrary example in particular would not be accepted by Atelier B due to a typing problem. It is nonetheless mathematically valid.

  This is patently a proof obligation displaying an error in the component that must be modified before proceeding. In the case of false proof obligation due to lack of information, think about:

  - a loop invariant that is too weak: let us remind that in a loop invariant, all variables used in the loop body must at least be typed:

– a linking invariant missing between an abstract variable and the imported variable(s) that realize it.

Proof obligations apparently meaningless can nonetheless be true by contradictory hypotheses. For example, in an IF..ELSIF without ELSE but whose case conjunction is still true can concern a variable that has not been initialized under the contradictory hypothesis that the case conjunction is false. This proof obligation is correct and detects no error in the component.

Chapter 6

Formal proof phase

This chapter is about the following key notions:

General method: use the Prover as much as possible.
Use `pr` or `pp` to complete a goal, `pr(Red)` to proceed.
Start in force 0 with an intiutive demonstration.

Commands are organized as: move, view various levels of proof commands
It is recommended to know all commands (about 35).
The Prover uses *inference rules* based upon *formula matching.*
Jokers are one-letter variables representing any formula. *Writing rules* enable to transform part of an expression
The *guards* enable to perform tests when applying a rule
User rules must be saved in the *pmm* file, they are loaded by `pc` and applied with `ar`.
The Predicate Prover operates by returning to quantified predicates: it is the `pp` command.

The Prover interface has several windows that can be juxtaposed and superposed.
The user can concentrate either on selecting a PO or on the proof.
The B component currently proved must be accessed easily.
The user must organize his/her screen with the Prover and the component currently proved.
The Interactive Prover can also be used in *batch* mode in a terminal.

The *command line* represents the proof tree - it is a key indicator.

There are four simple proof strategies:

- Using `pr` and `pp` ;
- Using Add Hypotheses and Prove by Cases;
- Search and use Prover rules;
- Using user rules.

Proof final check consists in replaying all demonstrations.
This check is done using the *Unprove* and Replay options in the proof menu
An *admission rule* enables to validate goals - to explore the remainder of the demonstration.
It is a rule added to the .pmm file, whose consequence is a joker The higher forces (1, 2, 3) are at times used in interactive proof.
Force 1 simplifies the hypotheses better, force 2 adds derived hypotheses.
The proof trace enables to analyze the action of a `pr` command.

There is a table displaying the proof command to be selected according to the type of situation.

Hypotheses must at times return to the Prover - to direct the Prover on these.
Equalities appearing as hypotheses, in particular, may request such an operation to be taken into account.
To write a hypothesis in the exact form enabling it to match with an item, use *AddHypothesis*.
Use *AddHypothesis* rather than a forward rule.
Use *AddHypothesis* to create correctly bracketed expressions.

Check the number of proof cases with `pr(Red)` rather than `pr`.
Do not delete user rules when this induces an offset in the rules numeration.
You must save your demonstration before changing forces during an interactive proof.

6.1 General method

Formal proof consists in demonstrating with the Interactive Prover the proof obligations remaining after the tuning-up phase (refer section 5). To perform efficient interactive proofs, you have to select the best from the automatic proof strategies, that is **use calls to the Prover as much as possible** . These calls are accessed through the *Prove* (`pr`) command that starts the Automatic Prover in current force. In some cases you can also try the *PredicateProver* (`pp`) command that can also automatically unload the goal.

The Automatic Prover is devised to get the best possible results in automatic proof - this is why it tries exploratory strategies before concluding on a failure. These strategies are harmful in interactive proof when they fail - as they determine the goal as a failure. Most times, they are proofs by cases, as in the following example:

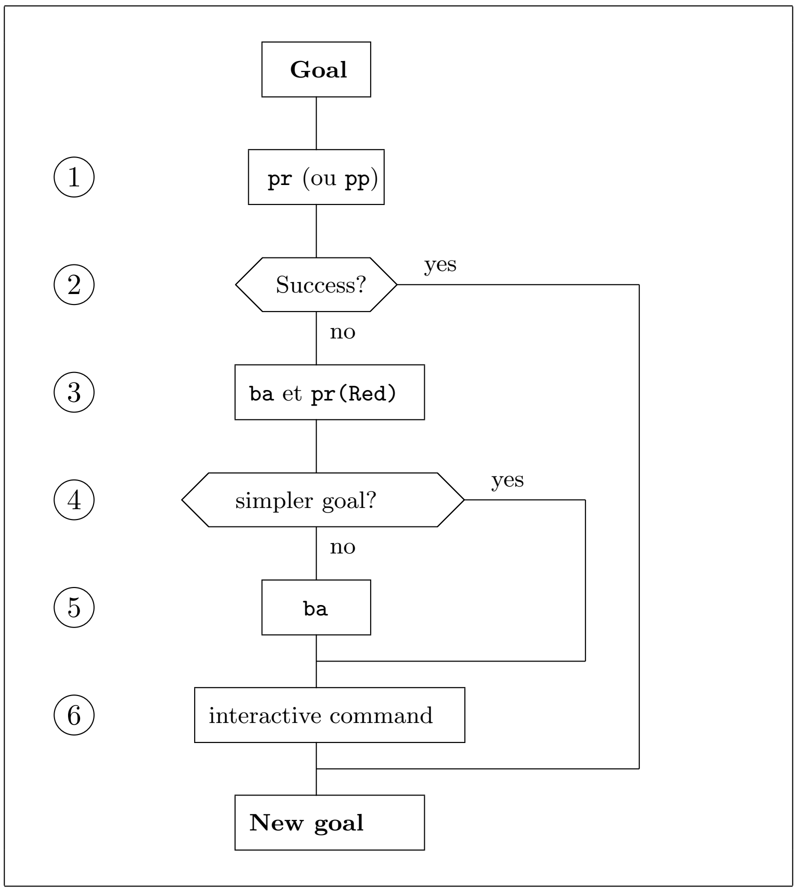
We have to prove a proof obligation of the form:

$H \ \wedge \ (p \ \Rightarrow \ q) \ \Rightarrow \ B$

When the Prover demonstration fails, it can decide to make two cases; p and$\neg p$ to try and use the $p \ \Rightarrow \ q$ hypothesis. If proof still fails, the failed goal can be the p one. The user proceeds with the interactive proof; when finished, only the p case will have been demonstrated, and the other case will have to be processed. This behavior is particularly annoying when the Prover tries proof by cases that are not useful for the demonstration - thus uselessly duplicating the proof. Proofs by cases are indicated only by case hypotheses, which do not raise the user's attention easily. This is why it is advised to use the `pr(Red)` command starting the Prover without the exploratory proof strategies, more precisely:

- launch `pr` ;
- if current proof path succeeds, proceed with the next goal;
- if proof fails, move back with *Back* (`ba`) and start `pr(Red)` to proceed with the proof without risking exploratory cases.

The general method to perform formal demonstrations is as follows:



This diagram explains how to proceed with the proof by processing each new goal. When none is left, proof is completed. Let us detail the steps:

Step 1 For each new step, start the Prover to loose no time if the step is automatically demonstrated. This step must be skipped when you are sure that the Prover cannot succeed; it is in particular the case with the starting goal. Its proof necessarily fails as the proof obligation is not automatically demonstrated. At this step, try also using the `pp` command - possibly preceded by `dd` to load hypotheses (we have seen in the previous chapter that the Predicate Prover reacts better when all hypotheses have been loaded).

Step 2 When the last proof path is unloaded, we get a new goal. Otherwise we go to step 3.

Step 3 The previous command can have tried exploratory proofs by cases. Thus we must return (`ba`) and use a reduced proof command to proceed in the proof safely.

Step 4 The previous command uses the automatic proof strategies and rules: the selected direction might not be the user desired one. At this step, we thus must decide if we will continue from the simplified goal with the reduced proof, or go back. In practice, it is enough to assess this new goal simply.

Step 5 Return when the simplified goal is not judged beneficial.

Step 6 Here is the most tricky step: we have to find the adequate interactive command to get the proof progress towards completion. The remaining part of this section deals with this problem.

For example, we have to demonstrate the following proof obligation:

$$\begin{array}{l} \text{"}\mathit{Local\ hypotheses}\text{"} \wedge \\ \mathit{ntt} \in \mathit{TACHES} \wedge \\ \neg(\mathit{ntt} \in \mathit{taches}) \wedge \\ \mathit{tt\$0} \in \mathit{taches} \cup \{\mathit{ntt}\} \wedge \\ \text{"}\mathit{Check\ that\ ...}\text{"} \\ \quad \Rightarrow \\ (\mathit{tconnait} \cup \{\mathit{ntt}\} \times \mathit{tconnait}[\{\mathit{tt}\}])[\{\mathit{tt\$0}\}] = \\ \quad \{\mathit{pass}((\mathit{tident} \lhd\!\!\!- \{\mathit{ntt} \mapsto \mathit{tident}(\mathit{tt})\})(\mathit{tt\$0}))\} \end{array}$$

We shall show only the local hypotheses on purpose: as shown above, this obligation is incomplete and thus non-demonstrable. But we are not reading hypotheses here - we usually have an intuitive demonstration stemming from a tuning-up phase. It is from this demonstration and not by reading the full obligation that we must start our formal proof (on this example, the reader will trust us for the intuitive demonstration).

We are in step 1. It is useless to try `pr` since the obligation is not automatically demonstrated. But the presence of unions and relations suggests us to try the Predicate Prover: `pp`, preceded by `dd` to separate hypotheses. The Predicate Prover fails. According to steps 2 and 3 we thus must return two steps backwards: `ba(2)` and call the Automatic Prover in reduced mode: `pr(Red)`. This call loads the hypotheses and creates derived hypotheses. The goal is not modified.

$$\begin{array}{l} (\mathit{tconnait} \cup \{\mathit{ntt}\} \times \mathit{tconnait}[\{\mathit{tt}\}])[\{\mathit{tt\$0}\}] = \\ \quad \{\mathit{pass}((\mathit{tident} \lhd\!\!\!- \{\mathit{ntt} \mapsto \mathit{tident}(\mathit{tt})\})(\mathit{tt\$0}))\} \end{array}$$

We are in step 4: the goal can be considered simpler as it no longer holds the hypotheses. We now move to step 6: selecting the interactive commands. The intuitive demonstration (the reader must trust us) suggests we make two cases: $\mathit{tt\$0} = \mathit{ntt}$ or $\mathit{tt\$0} \neq \mathit{ntt}$. The command is thus:

```
dc(tt$0 = ntt)
```

The goal becomes :

$$\begin{array}{l} \mathit{tt\$0} = \mathit{ntt} \;\Rightarrow \\ (\mathit{tconnait} \cup \{\mathit{ntt}\} \times \mathit{tconnait}[\{\mathit{tt}\}])[\{\mathit{tt\$0}\}] = \\ \quad \{\mathit{pass}((\mathit{tident} \lhd\!\!\!- \{\mathit{ntt} \mapsto \mathit{tident}(\mathit{tt})\})(\mathit{tt\$0}))\} \end{array}$$

We are back in step 1: we must try `pr` that does not always demonstrate the current goal. We could then try `pp`, but the Predicate Prover is not frequently used within a proof as it takes a minute to fail. We perform thus step 2: `ba` and `pr(Red)`. The goal becomes:

$$(\mathit{tconnait} \cup \{\mathit{ntt}\} \times \mathit{tconnait}[\{\mathit{tt}\}])[\{\mathit{ntt}\}] = \{\mathit{pass}(\mathit{tident}(\mathit{tt}))\}$$

We are in step 4. As the goal is much simplified, we proceed to step 6: selecting a new command. Here, the new goal still holds unions and relations; the Predicate Prover might demonstrate this. The selected command is pp and the goal becomes:

$$\begin{array}{l} tt\$0 \neq ntt \;\Rightarrow \\ (tconnait \cup \{ntt\} \times tconnait[\{tt\}])[\{tt\$0\}] = \\ \quad \{pass((tident \lhd \{ntt \mapsto tident(tt)\})(tt\$0))\} \end{array}$$

We are now in the second case that is similarly demonstrated (we shall see in this chapter which case we are up to). We perform the two previous steps again, the proof succeeds. The proof tree is as follows:

```
Force(0) &
  pr(Red) &
    dc(tt$0 = ntt) &
      pr(Red) &
        pp &
      pr(Red) &
        pp &
  Next
```

This tree is displayed in one of the areas of the MOTIF interface - as we are going to see. The proof is completed in 6 steps. We see from this example to what extent we can rely on pr and pp to unfreeze a demonstration by using the actual flow-chart given in this section. Additionally, you will consider the following points:

- **The interactive proof is used in force 0**. It is always possible to use higher forces but this is seldom justified: remember that it takes 1 minute to load 100 hypotheses in force 1 against less than 1 second in force 0. Performance gains in an interactive proof - that is, guided by the user - does not justify such a cost. A PO is accessed in interactive mode by resetting it to 0 using gr (GotowithReset). To return to force 0 on a loaded proof obligation, simply use the ff(0) command. Refer to section 6.8.3.
- **Start with an intuitive demonstration**. When tuning-up, the concerned PO must have been visited, a justification and an intuitive demonstration have been made. We will start from this state. In principle, notes have been taken with this in mind when tuning-up. The intuitive reasoning will be repeated.

The remainder of this chapter is divided in the following way:

- Introduction to the interactive proof: the key notions to use this tool are explained there. In particular, commands are given by order of importance.
- Using the interface: the Interactive Prover operates with a MOTIF interface - its use is described in the present section.
- The command line: this key landmark for the proof is also detailed here.
- Simple proof strategies.

- Advanced use of the Prover.
- Proof tricks: you will find the table of commands to select according to each type of situation.
- Traps to be avoided: this section must be read before any intensive use of the Prover.

6.2 Introduction to the interactive proof

This part is aimed at users discovering the interactive proof but already familiar with B language, mathematical proof principles and notations used in proof obligations. The notions described are as follows:

- The Interactive Prover commands;
- The rule language;
- The Predicate Prover.

6.2.1 The Interactive Prover commands

The following explanations do not replace the Interactive Prover Reference Manual. In particular, the exact syntax for each command is not described. In this section, we want to give the necessary indications to think about and use the adequate command at the right time. If you do not know the Interactive Prover commands, you are recommended to read the present section.

We have distributed the different commands under various headings and ordered these by order of importance for a good use of the Prover. These headings are:

- move commands;
- reading commands;
- automatic proof commands;
- proof commands with no added rules;
- commands to add protected rules;
- user rules;
- generalizing;
- proof commands for specific cases;
- other commands.

Let us now detail these headings.

Move commands

The following commands are used to move within proofs. It is difficult to assign them an order of importance as they are all required to use the Interactive Prover.

- **Goto** (`go`): used to position in a proof obligation. With the Motif interface, it is usually simple to double-click the PO in the list; thus this command is mainly used in batch mode (refer 6.3.2 section). In this mode, it is absolutely required.
- **Next** (`ne`): moves to the next unproved PO.
- **Back** (`ba`): cancels the previous interactive command that has affected the proof. It is much used after a failed call to the Automatic Prover - according to the method described at the beginning of this chapter. **Use Back to delete useless commands or to place an interactive command before current action**.
- **GotowithReset** (`gr`): use this command to go to a proof obligation and reset its force to 0. When there are many hypotheses, loading a PO can be lengthy, specially for an operation first PO where it includes the loading of all context hypotheses (1 minute per hundred hypotheses in force 0. **Use GotowithReset when there are several hundred hypotheses to reach an operation first PO**.
- **SaveWithoutquestion** (`sw`): requests the forced backup of the command line.. **Use `sw` from time to time when the demonstration becomes long, specially before changing forces or during repeated backward moves**.
- **Reset** (`re`): used to return to the beginning of a PO demonstration, either to read an initial goal (think also to `lp`), or to return to a demonstration with the experience of the previous test (think to save before, `sw`).
- **Step** (`st`) : the *Step* command executes the next saved command and advance by one step the cursor in the saved demonstration. **Use Step to replay a saved proof**.
- **Force** (`ff`): used to replay a demonstration in a different force. Frequently used to reset to zero a PO force before starting an interactive demonstration. This command should be reserved to the rarest cases where the user wants to attempt using a higher force (refer to section 6.8.3). Otherwise use instead *GotowithReset*.
- **Repeat** (`rr`): repeats the last entered command.
- **Quit** (`qu`): quits the Interactive Prover. With the Motif interface, the user can also use the specific buttons.

reading commands

The two following commands enable to read proof obligations.

- **SearchHypothesis** (`sh`): this information command is used to search all hypotheses verifying a specified filter. Its most frequent use is to search all hypotheses affecting a variable. **Use SearchHypothesis as soon as the search for hypotheses turns difficult**.
- **showReducedPo** (`rp`): this command displays a reduced proof obligation, that is reduced to the sole hypotheses sharing a common symbol with the goal. This is what has to be examined first to check rapidly if a PO or sub-goal are true. . **Use showReducedPo to find an intuitive demonstration or in conjunction with SearchHypothesis**.

Automatic proof commands

We now describe the two automatic proof commands:

- **Prove** (`pr`): is a call to the Automatic Prover - its use must be controlled, specially for parasite proofs by cases (refer section 6.8.1). But you must never demonstrate by yourself what the Automatic Prover is able to demonstrate. **Never forget to try a call to the Automatic Prover on a goal before entering interactive commands to demonstrate it**.
- **PredicateProver** (`pp`): this command calls the Predicate Prover on current lemma. The Predicate Prover (refer to section 6.2.4) is often successful on goals made up of assembled or functional expressions. **Think to use PredicateProver when the goal is made up only of assembled or functional expressions**.

Proof commands with no added rules

The following commands are used to succeed with proofs without adding user rules or rules controlled by the Predicate Prover.

- **AddHypothesis** (`ah`): the user can propose new hypotheses to be added to the existing ones after a demonstration of the proposed hypothesis. This command thus enables to add supererogatory hypotheses. These will be entered by the user who can thus introduce additional information in the proof by bringing in the benefits of his/her intuition and imagination. *AddHypothesis* is one of the most important proof commands. It enables to direct the proof by loading intermediate goals starting from the hypotheses to reach the main goal. An other use is to have the hypotheses entered in the Prover - refer to section efepihyp. **Think to use AddHypothesis as soon as a new expression is needed or when hypothesis is not completely simplified**.
- **Deduction** (`dd`): this command loads the h hypothesis when the goal has the form: $h \Rightarrow B$, with no intervention of the Automatic Prover. **Think to use Deduction when the Prover loads a hypothesis under a non-favorable normalized form**. But a hypothesis introduced in such a way is not processed by some of the Prover mechanisms, notably the equality simplifier.

- **DoCases** (`dc`): This command enables to prove by cases. The user can submit a p predicate as an argument - the proof is then performed in the p and$\neg p$ cases - or he/she can give the name of an v variable and a E finite set of small order - the proof then proceeds for v being each of E elements. In this last case, we first must prove $v \in E$. Many demonstrations can only be performed by cases, which the Prover does not always detect. **Use DoCases when the intuitive demonstration is done by cases**.
- **SearchRule** (`sr`): used to search for a rule in the Automatic Prover rule database. **Use SearchRule before attempting to add a rule**. If hypotheses are missing to apply the encountered rule, use `ah` instead of writing a replacement user rule.
- **ApplyRule** (`ar`): this command is used to apply a rule from the Prover database found by *SearchRule*. **Use ApplyRule as soon as it is clear that a trivial mathematical rule enables to conclude**. You must indeed try and apply the simplest possible rules, these rules being thus more likely to be in the database.

Commands for user rules

The following commands are required when you wish to use user rules:

- **ApplyRule** (`ar`): this command enables to apply a rule. It can be a rule from the database and found by *SearchRule*, a rule added after proof by the Predicate Prover (refer section 6.2.4) or a user rule. **Remember to use ApplyRule** as soon as it is clear that a trivial mathematical rule enables to conclude. You must indeed try to apply the simplest possible rules, these rules being thus more likely to be in the database or to be demonstrated by the Predicate Prover. If the rule must be added by the user, it will be more easily validated if it is simple .
- **PmmCompile** (`pc`): command used to read and compile the `component.pmm file` storing the user rules. **Use PmmCompile to take into account a modification in the .pmm file**.

Generalizing The following command is used when a first proof obligation is successful and when the user wishes to generalize his/her demonstration to other obligations.

- **TryEverywhere** (`te`): trying to generalize a demonstration. Enables to execute the proof of a set of proof obligations by modifying or replacing their command lines from a sequence of commands given as parameters. **Use TryEverywhere after a successful demonstration if other POs seem to be similarly proven.**.

Special cases

The following commands are applied in special proof cases.

- **useEqualityinHypothesis** (`eh`): this command enables to replace a with A in the goal or in the hypotheses when $a = A$ or when $A = a$ is an hypothesis. It is useful when the Prover does not perform a desired replacement, when the equality is a derived hypothesis (refer to this notion in section 6.7.2) or if we want to rewrite a hypothesis preceding the equality and thus not modified by it. **Remember to use useEqualityinHypothesis as soon as an equality must be used in a goal or a hypothesis**.

- **SuggestforExist** (`se`): when a goal has the form: $\exists x.P(x)$, the user can propose x_0 values for the x variables. The goal thus becomes $P(x_0)$. When demonstrated, the initial existential goal is established, as we have shown a value verifying the predicate. Practically, in 90

- **FalseHypothesis** (`fh`): when one of the hypotheses contradicts the others, the goal has no interest: it is enough to show that the current hypotheses establish the negation of this hypothesis. The user can then point to an hypothesis assumed to be contradictory, the goal now becoming the negation of this hypothesis. It is not possible to remove the concerned hypothesis so that it appears only in the goal. The `fh` command does not perform it thus: but this is not a problem: the resulting lemma is simply true both by the goal and by contradictory hypotheses. **Remember to use FalseHypothesis when the proof of a PO true by contradictory hypotheses fails** .

- **Contradiction** (`ct`): used for the proof by contradiction. The goal negation is loaded as hypothesis and the goal becomes *bfalse*. This is specially useful when the goal has the form: `not(p)`, as it is then p that is loaded as hypothesis and the presence of *bfalse* as a goal indicates to the Automatic Prover mechanisms to search for a contradiction. Additionally, if the goal has the form: `not(a = b)`, rewriting a as b induced by the $a = b$ hypothesis can very easily show the contradiction. **Remember to use Contradiction when the goal has the form: `not(p)`, and specially the form: `not(a = b)`.**

Other commands

Lastly, the following commands are more seldom used.

- **ParticularizeHypothesis** (`ph`): for a hypothesis of the form: $\forall x.(P(x) \Rightarrow Q(x))$, the user can propose a list of x_0 values; he/she will then have to prove the $P(x_0)$ subb-goal for the $Q(x_0)$ hypothesis to appear. **Remember to use ParticularizeHypothesis when the Automatic Prover cannot instantiate by itself a $\forall x.(P(x) \Rightarrow Q(x))$ hypothesis**. When, in particular $P(x_0)$ is not directly a hypothesis, to prove it as a sub-goal enables to use all the Prover functionalities to display it.

- **ModusponensonHypothesis** (`mh`): used to generate Q when $P \Rightarrow Q$ and P are hypotheses. The Automatic Prover does this systematically but for already derived hypotheses (refer section 6.7.2).

- **showLitteralPo** (`lp`): this command displays a proof obligation as specified in the component file without even having to load the concerned PO. This is at times useful to rapidly find the link between a proof obligation and the component source.

- **GlobalSituation** (`gs`): this command returns a list of all proof obligations with their status and goal. With the Motif interface, the user should not have to use it as it is useful only in batch mode (refer section 6.3.2).
- **GotoWithoutsave** (`gw`): this seldom used command enables to quit without saving a proof obligation.
- **SavewithQuestion** (`sq`): saves current command line asking the user for confirmation when it is not evident that the replacement of the saved demonstration with the current demonstration is beneficial (for example: when neither of the two demonstrations succeeds).

6.2.2 Rules and their use

A formal mathematical proof is performed by using mathematical rules. For example we demonstrate:

$p \in s \leftrightarrow t \wedge q \in s \leftrightarrow t \Rightarrow \mathsf{dom}(p) - \mathsf{dom}(q) \subseteq \mathsf{dom}(p - q)$

by using the definition of the inclusion:

$\mathsf{dom}(p) - \mathsf{dom}(q) \subseteq \mathsf{dom}(p - q) \Leftrightarrow \forall x.(x \in \mathsf{dom}(p) - \mathsf{dom}(q) \Rightarrow x \in \mathsf{dom}(p - q))$

then the definition of a relation domain, etc. The mathematician often performs demonstrations without explicitly quoting all rules he/she uses. But the selection of the rules considered as permitted is important - to know what is called a correct demonstration. It would be too easy to claim that the lemma to be demonstrated is always a permitted rule. In a B demonstration, the permitted rules are those from the language construction theory (refer to B-Book, [**?**]) and those from the Prover database. How can such mathematical rules be converted into something executable on a computer? We will introduce a few concepts to understand this point. These are the bases of the theory language on which is built Atelier B Prover. The theory language is implemented by a software layer called *Logic Solver*. For the prover user, it is sufficient to know the few notions hereafter given to be able to find a rule in the math rule or write a user rule. The theory language is fully specified in the Logic Solver Reference Manual [**?**].

Formula matching concept

Let us explain this notion with an example. We say that the formula:

$aa + (bb - cc) \times 12$

matches the template:

$x + y \times z$

because replacing x with aa, y with $bb - cc$ and z with 12 enables to get the formula from the template. In a template, one-letter identifiers have a specific function: they replace any formula. They are known as **jokers**. Thus $aa + bb$ matches $aa + x$ or $u + v$, but not

$x + aa$. Beware also the implicit brackets: $aa + bb + cc$ matches $x + y$ (x on $aa + bb$ and$y$ on cc) as $aa + bb + cc$ is understood as $((aa + bb) + cc)$ [1].

Inference rule notion

Let us explain this notion with an example; if the goal is to prove:

$\mathsf{bool}(xx - 2 \leq 10) = \mathsf{bool}(xx \leq 12)$

then the following inference rule can apply:

$(a \;\Rightarrow\; b) \;\wedge$
$(b \;\Rightarrow\; a) \;\wedge$
$\Rightarrow$
$\mathsf{bool}(a) = \mathsf{bool}(b)$

as the goal matches the $\mathsf{bool}(a) = \mathsf{bool}(b)$ template, with the following filter:

$a \;\rightsquigarrow\; xx - 2 \leq 10$
$b \;\rightsquigarrow\; xx \leq 12$

This $\mathsf{bool}(a) = \mathsf{bool}(b)$ template, placed after the imply symbol is called a <u>consequence</u> of the rule, when $(a \;\Rightarrow\; b)$ and $(b \;\Rightarrow\; a)$ are the <u>antecedents</u>. Application of this rule triggers an unloading of the initial goal and a display of the following sub-goal:

$a \;\Rightarrow\; b$ that becomes devient $(xx - 2 \leq 10 \;\Rightarrow\; xx \leq 12)$
$b \;\Rightarrow\; a$ that becomes $(xx \leq 12 \;\Rightarrow\; xx - 2 \leq 10)$

Some rules have no antecedents; in this case the symbol imply is omitted. For example:

$\exists x.(x \in \mathsf{BOOL})$

These rules are called sheet or terminal rules as they delete a goal without creating a new one. A proof succeeds when all goals have disappeared through the sheet rules action.

Rewriting notions

A rule whose consequence has the form $E == F$ is a <u>rewriting</u>, it does not operate as a standard rule. It applies when a sub-formula exists in the goal that matches the E template. The new goal is thus produced by a replacement of E with F. For example, the rule:

$\neg\neg P == P$

transforms $\{uu \mid uu \in \mathbb{N} \;\wedge\; \underline{\neg\neg(uu = 0)}\} = \{0\}$ en $\{uu \mid uu \in \mathbb{N} \;\wedge\; \underline{uu = 0}\} = \{0\}$. The sub-formula $\neg\neg(uu = 0)$ is localized and transformed into $uu = 0$.

[1]Brackets in a formula are only associativity modifiers; we for example see no difference between $3 + (3 \times 6)$ and$3 + 3 \times 6$.

Each application of a rewriting rule transforms only one occurrence of the left sub-formula. Thus:

$aa == bb$

transforms the $aa + aa = 0$ into $aa + bb = 0$ for the first application of the rule then into $bb+bb = 0$ for the second one (we shall not explain here why it is the right side occurrence that is first transformed as this would lead us too far). When applying a rule through *ApplyRule*, a rewriting rule always re-applies as long as it transforms something (*Once* and *Multi* modes do not apply to rewriting). **Beware of the loops**: the rule

$x == x + 0$

induces a loop. For example on the $aa = bb$ goal:

$aa = bb$
$aa = bb + 0$
$aa = bb + 0 + 0$
$aa = bb + 0 + 0 + 0$
etc.

Let us note that this rule is additionally false, as x can match non-numerical expressions.

Notion of guards

It often happens that a rule to be applied needs more information than provided by the rule. For example, we have to prove:

$xx \leq yy \ \wedge$
$yy \leq 5$
$\Rightarrow$
$xx \leq 5$

Which inference rule would allow to unload this lemma? The following rule is not applicable:

$x \leq y \ \wedge$
$y \leq z$
$\Rightarrow$
$x \leq z$

Let us analyze how it would apply:

the $xx \leq 5$ goal matches with $x \leq z$ with the filter :
$x \leadsto xx$
$z \leadsto 5$
New goals are produced:
$x \leq y$ produces $xx \leq y$
$y \leq z$ produces $y \leq 5$

We note that as the y joker is not instanced, it remains unchanged in these new goals. Alas, no variable in the lemma to be demonstrated is called y in a single letter; these goals are thus meaningless as they use an undefined variable. The considered rule is not suitable to demonstrate this lemma.

To solve this case, we have to be able to write rules able to search the hypotheses directly : specific functionalities of the theory language called guards. The theory language offers about thirty guards - most of which are not useful in the framework of the interactive proof. We shall limit ourselves to the guards described hereafter. The principle of a guard is to perform a computer process after the goal has matched the rule consequence but before the production of new goals. This process can have different results but it always returns TRUE or FALSE. This rule applies only when the result is TRUE.

The list of guards useful for the proof is as follows:

- **binhyp**: to search for a hypothesis. E. g.:

  $\mathsf{binhyp}(x \leq y)\ \wedge$
  $y \leq z$
  $\Rightarrow$
  $x \leq z$

  This rule applies on the following example:

  $xx \leq yy\ \wedge$
  $yy \leq 5$
  $\Rightarrow$
  $xx \leq 5$

  First, the $xx \leq 5$ goal matches $x \leq z$, x and z are instanced on xx and on 5. Then the search for $xx \leq y$ is performed by moving upwards in the hypotheses: these are $yy \leq 5$ and $xx \leq yy$. Matching is then produced for $xx \leq yy$ by instanciating y on yy, the guard is TRUE. The $yy \leq 5$ is then produced. Caution: the search for a hypothesis stops at the first match. When the lemma is:

  $xx \leq yy\ \wedge$
  $xx \leq uu\ \wedge$
  $yy \leq 5$
  $\Rightarrow$
  $xx \leq 5$

  The rule always applies, but the goal produced is $uu \leq 5$. This does not allow the proof to succeed.

- **band** : this is a guard cut-on. $\mathsf{band}(g_1, g_2)$ triggers the execution of the g_1 guard for all possible cases as long as g_2 is not verified. For example, the rule

  $\mathsf{band}(\mathsf{binhyp}(x \leq y), \mathsf{binhyp}(y \leq z))$
  $\Rightarrow$
  $x \leq z$

applies to the following example:

$$\begin{aligned}&xx \leq yy \ \wedge\\&xx \leq uu \ \wedge\\&yy \leq 5\\&\Rightarrow\\&xx \leq 5\end{aligned}$$

First, the $xx \leq 5$ goal matches $x \leq z$, x and z are instantiated on xx and on 5. Then the search for $xx \leq y$ is performed by moving upwards in the hypotheses: the match is produced for $xx \leq uu$ by instantiating y on uu, the first binhyp is TRUE. But there is no $uu \leq 5$ hypothesis that would allow the success of the second binhyp. Through the action of band the upward run-around from the first binhyp proceeds, the match occurs then on $xx \leq yy$ enabling to succeed.

- **btest**: this guard evaluates a literal expression of one of the following expressions: $a = b$, $a \neq b$, $a < b$, $a > b$, $a \leq b$ and $a \geq b$. The corresponding ASCII writing is:

```
btest(a = b)
btest(a /= b)
btest(a< b)
btest(a> b)
btest(a <= b)
btest(a >= b)
```

The last four forms that concern the order relation are TRUE only if a et b literal and positive integers. E. g. the $btest(x > 0) \Rightarrow x \in \mathbb{N}$ rule enables to unload the $3 \in \mathbb{N}$, $100 \in \mathbb{N}$ goals, etc. The $btest(a = b)$ and $btest(a \neq b)$ forms concern lexical equality, they test if a et b are lexically true. For example, $btest(var1 = var2)$ is false, even if there exists an hypotheses that states that $var1 = var2$, but $btest(aa = aa)$ is true. Caution, the lexical equality applies only to integers and indentifers. E. g.: $btest(xx + yy = xx + yy)$ is false as $xx + yy$ is neither an integer nor an identifier.

6.2.3 Writing a user rules file

As a last resort, the user can introduce an undemonstrated user rule to fix a proof. This method is to be used only after the failure of the proof with the interactive commands and failure of adding the desired rule as a rule protected par the Predicate Prover (*AddUserrule command*). The rule is then written in a file in theory language. This file must be named `component.pmm` (pmm for Proof Methods Manual), o- `component` is the component name without extension: it must be placed in the `bdp` directory that corresponds to the project. Atelier B does not create this file by default; you can thus easily make sure that there are no user rules for a component.

The syntax to be respected is as follows:

- place the rules in the theories declared by `THEORY name IS list of rules END` ;
- separate rules with semi-colons;

- separate the theories with `&`.

Here is an example of a rules file:

```
THEORY MyRules IS

  binhyp(a<=b)
  =>
  b+1-1: a..b;

  a<=b
  =>
  a-1-b+1<=0

END

&

THEORY NatCalc IS

  a: NATURAL &
  b: NATURAL
  =>
  a+b: NATURAL

END
```

We note that the rules in this file are not necessarily equivalencies. As an example, let us examine the following rule:

```
a: NATURAL &
b: NATURAL
=>
a+b: NATURAL
```

This rule is not an equivalence. When applied on the $1 + 2 \in \mathbb{N}$ goal, it does produce the two true goals, $1 \in \mathbb{N}$ and $2 \in \mathbb{N}$ implying the initial goal - but on the $1 + (-2) \in \mathbb{N}$ goal, this rule produces the false goal $(-2) \in \mathbb{N}$ and provokes the failure of the proof. Thus use wisely such rules.

The user rules file for a component is not automatically read by the Prover after each modification: that would be too heavy. For the modifications to be taken into account, we must use the *PmmCompile* (`pc`) command. We are now to give a full utilization example:

We have to demonstrate the following lemma: *"Local hypotheses"* $\wedge$
$cc \in \mathbb{N} \wedge$
$cc \leq 2147483647 \wedge$
"Check that ..."

$\Rightarrow$
$cc \bmod 3 \in 0 \,..\, 100$

The Automatic Prover has no rules concerning the modulo - we can notice this with *SearchRule* :

```
PRI > sr(All, (a mod b))
Searching in All rules with filter
    consequent should contain a mod b Starting search...
Found 0 rule(s) for this filter.
```

It is thus clear that it cannot demonstrate this lemma. The Predicate Prover (*PredicateProver*, `pp` command) also fails. The only way to solve this problem is to add a rule that will introduce the missing mathematical knowledge. To add the rule we simply open the *component .pmm* file in the project database directory . You can edit this file with your favorite editor or select the `Edit PO methods` option in the `Show/Print` menu in the Interactive Prover global situation window. The selection opens the rules file in edit mode corresponding to the component currently being proved - additionally, this file is created when missing.

In this rules file we enter:

```
THEORY ModProps IS

  x: NATURAL
  =>
  x mod 3 : 0..2

END
```

We then use the *PmmCompile* (`pc`) command to load the file in the Prover:

```
PRI > pc
Loading theory ModProps
```

After loading the local hypotheses with a *Deduction* (`dd`) command, we are to use the added rule in our demonstration. This rule does not fit for the $cc \bmod 3 \in 0 \,..\, 100$ goal, it would fit for $cc \bmod 3 \in 0 \,..\, 2$. We are therefore to add this last expression as a new hypothesis (*AddHypothesis*, `ah` command). The Prover asks us to demonstrate it - which we can do using the added rule. and the *ApplyRule* (`ar`) command. The rule produces the $cc \in \mathbb{N}$ goal that the Interactive Prover proves directly (`pr` command). The new hypothesis is then accepted and the goal becomes:
$cc \bmod 3 \in 0 \,..\, 2 \;\Rightarrow\; cc \bmod 3 \in 0 \,..\, 100$
This last goal is directly demonstrated by the Automatic Prover and the demonstration completed. This demonstration process is given below as it is displayed in the Interactive Prover command window:

```
PRI > ah(cc mod 3 : 0..2)
Starting Add Hypothesis
```

```
Current PO is Initialisation.1
    Unproved saved Unproved
    Goal
        cc mod 3: 0..2
End
PRI > ar(ModProps.1,Once)
Starting Apply Rule
Current PO is Initialisation.1
    Unproved saved Unproved
    Goal
        cc: NATURAL
End
PRI > pr
Starting Prover Call
Current PO is Initialisation.1
    Unproved saved Unproved
    Goal
        cc mod 3: 0..2 => cc mod 3: 0..100
End
PRI > pr
Starting Prover Call
Current PO is Initialisation.1
    Proved saved Unproved
    Goal
        ...
End
```

The *ApplyRule* command has been used with the following syntax: `ar(ModProps.1,Once)`, which means applying rule 1 from the *ModProps* theory only once: (*Once* meaning *just that*). This the most frequent use of the *ApplyRule* command, but for the rewriting rules where `ar(Theory.number, Goal)` must be used to specify that the rewriting must occur in the goal. For more details on *ApplyRule* refer to the Prover Reference Manual [**?**].

6.2.4 The Predicate Prover

The mechanism on which Atelier B main Prover is built is rule inference. This means that a demonstration will succeed only if rules are linked in order to have all goals disappear. There is no theoretical completion guarantee - and the equilibrium of rules between themselves is reached through empirical settings.

Other proof methods exist. One of these consists in bringing back a proof in the quantified predicates basic language then trying several instantiations to get a proof by contradiction. For example, this is the demonstration of $x \in A \ \wedge \ A \subseteq B \ \Rightarrow \ x \in B$:

search for a contradiction in:
$x \in A \wedge$
$A \subseteq B \wedge$
$\neg(x \in B)$
Transforming into quantified predicates:
$x \in A \wedge$

$\forall y.(y \in A \Rightarrow y \in B) \land$
$\neg(x \in B)$
Instantiation by x the second hypothesis:
$x \in B$
That contradicts the last hypothesis.

The performances of this method can be foreseen in this example: chances of success are high even if the demonstration is mathematically complex - but a limited number of hypotheses is required to avoid saturation problems associated with the important expansion during the transformation into quantified predicates. The possible instantiation choices are many, which bring proof delays, at times lengthy. On the other hand, this method is not adequate for arithmetic demonstrations.

Atelier B Predicate Prover is based on this principle; it has enabled to validate the main Prover mathematical database. It is used through the *PredicateProver* we are now to study.

The PredicateProver command

When in interactive proof, it is always possible to try and prove a proof path. The *PredicateProver* (`pp`) starts the Predicate Prover on current lemma - possibly trimmed of some hypotheses and with a maximum delay. We recommend the following use:

If the lemma can be demonstrated without arithmetic notions:		
If there are few hypotheses	`pp`	proof on the complete lemma, 60 maximum
If the required hypotheses are those with a symbol in common with the goal	`pp(rp.1)`	proof on the reduced lemma, 60 maximum
There is a large number of hypotheses but the required ones are few	returns in the goal the interesting hypotheses using the `ah` command and use `pp(rp.0)`	proof on lemma with selected hypotheses, 60 maximum 60s

As we have seen since the beginning of this chapter, the `pp` command is used alone or within a demonstration using other commands. Here is a short example of *PredicateProver* use with reduced hypotheses.

We have to demonstrate the following lemma:

$EE \in \mathbb{F}(\mathbb{Z}) \land$
$\neg(EE = \varnothing) \land$
$cc \in EE$
$\Rightarrow$
$\mathsf{card}(EE \cup \{cc\}) = \mathsf{card}(EE)$

According to the key principle governing the Interactive Prover use, we start by calling the Automatic Prover in reduced mode then examine the remaining goal:

```
PRI > pr(Red)
Starting Prover Call
```

```
Current PO is Initialisation.2
    Unproved saved Unproved
    Goal
        card(EE)+1-1 = card(EE)
End
```

Why does the Prover fail on this goal? In fact it is a problem of brackets - that is the $\mathsf{card}(EE) + 1 - 1$ analyzed as the $(\mathsf{card}(EE) + 1) - 1$ expression, thus a lack of simplification. Without further analysis, we doubt that submitting such a goal to the Predicate Prover will induce its simplification during the more complete analysis producing a transformation into quantified predicates. There might be many hypotheses - in this case the `pp` command can be lengthy as all hypotheses have to be converted. No hypothesis is required to demonstrate this goal, thus we can use `pp(rp.0)`.

```
PRI > pp(rp.0)
Starting Prover Predicate Call
Proved by the Predicate Prover
Current PO is Initialisation.2
    Proved saved Unproved
    Goal
        "‘Check that the invariant ...’" => card(EE\/{cc}) = card(EE)
End
```

The demonstration is completed, the goal area turns green and current state is “Proved”.

6.2.5 Protecting user rules

Since the Predicate Prover is well suited to rule proof, it is normal to use it to demonstrate user rules added by the user.

This has been implemented with the *Validation of Rule* (`vr`) command which allows to demonstrate a rule with the predicate prover. When the user adds only this type of user rules, the proof is valid without further verification. Let’s see this principle on an example.

We have to demonstrate the following lemma:

$$\begin{array}{l} 22 \in 20 \,..\, 30 \\ \Rightarrow \\ 22 \in 1 \,..\, 2 \;\vee\; 22 \in 20 \,..\, 30 \;\vee\; 22 \in 300 \,..\, 400 \end{array}$$

This is indeed evident. This lemma nonetheless fails in force 0 as there is no mechanism to acknowledge a hypothesis in a disjunctive goal. How can we eliminate this easy goal? A simple solution is to add a rule protected by the *Valid of Rule* (`vr`) command. We can add the rule $b \;\Rightarrow\; a \;\vee\; b \;\vee\; c$, for example in the NonEqui theory (the rule is, for instance, NonEqui.1), in the pmm file of the component and try to demonstrate it during an interactive proof session:

```
PRI > vr(Back, (b => (a or b or c)))
The rule was proved
```

In the above command, the `Back` keyword indicates it is a standard inference rule – as opposed to 'forward' rules. The second parameter is the rule statement. The Prover returns that the rule has been successfuly demonstrated by the Predicate Prover. This rule can then be used with a *ApplyRule* command:

```
PRI > ar(NonEqui.1,Once)
Starting Apply Rule
Current PO is Initialisation.1
    Unproved saved Unproved
    Goal
        22: 20..30
End
```

A single call to the Automatic Prover enables to complete the demonstration. No additional verification is required on this added rule: it has been demonstrated by the Predicate Prover.

6.3 Using the Interactive Prover interface

The Interactive Prover interface allows for a multi-window display of the proof. We will examine how to use it.

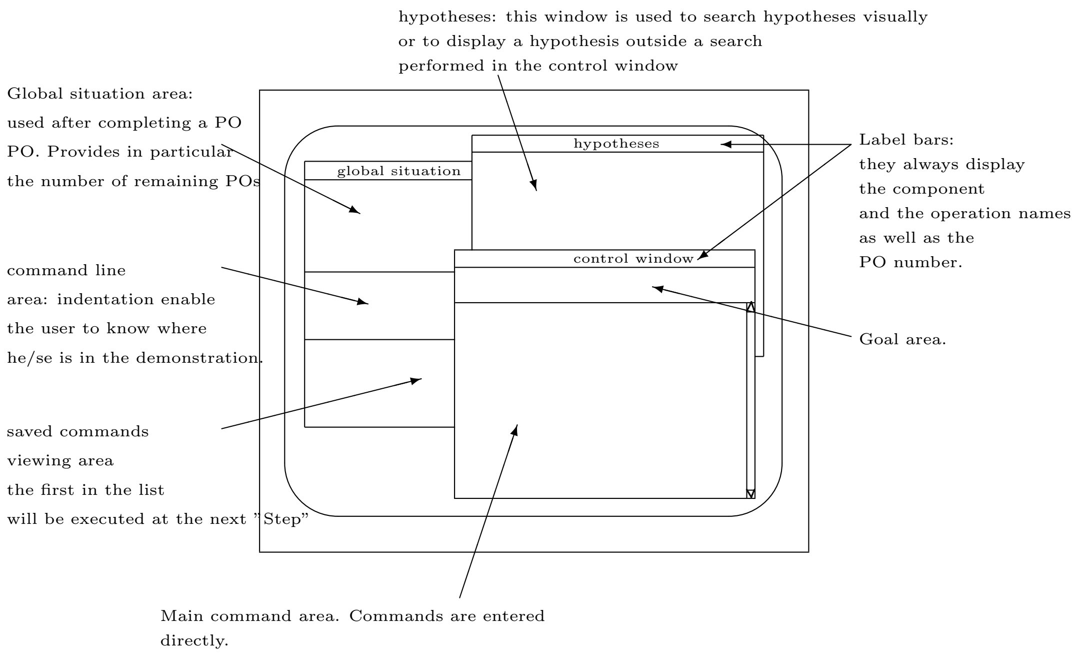
6.3.1 Screen organization

The Interactive Prover Motif interface was designed to facilitate the user tasks, in a multi-windows way. This means that each item requiring attention from the user is represented by a window, the two or three items being processed line by line on screen and the environment items iconized on the screen sides. Lining thus two windows that can overflow on the other enables to display more information on screen than would fit into its size, the Front / Back function being used to immediately place the window being read on top.

This multi-window pattern requires a adequate installation of the system. The following commands must be directly accessible:

- iconize or uniconize a window from any part of its area;
- move a window front or back from any part of its area.

The Interactive Prover standard position is as follows:

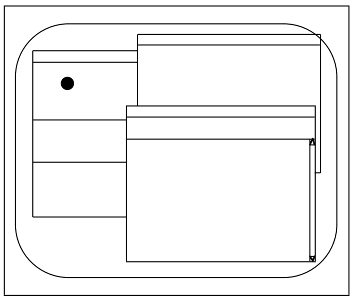

hypotheses: this window is used to search hypotheses visually
or to display a hypothesis outside a search
performed in the control window
Global situation area:
used after completing a PO
PO. Provides in particular
the number of remaining POs
global situation
hypotheses
Label bars:
they always display
the component
and the operation names
as well as the
PO number.
control window
command line
area: indentation enable
the user to know where
he/se is in the demonstration.
Goal area.
saved commands
viewing area
the first in the list
will be executed at the next "Step"
Main command area. Commands are entered
directly.

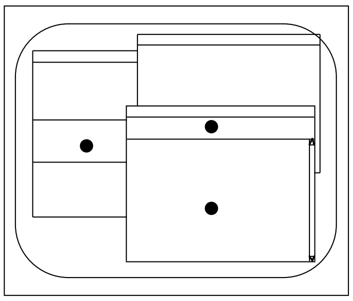
There are two main ways to use the screen in a proof session:

- **moving within the POs**: when the user has completed the demonstration, he/she checks what is the next proof obligation to be processed and how many are left. The user then focuses his/her attention on the global situation window.

- **proof**: when the user performs a proof, his/her attention is focused on the goal, the command and the command line areas enabling him/her to select a location in the demonstration.

Focusing areas in these two modes are as follows:

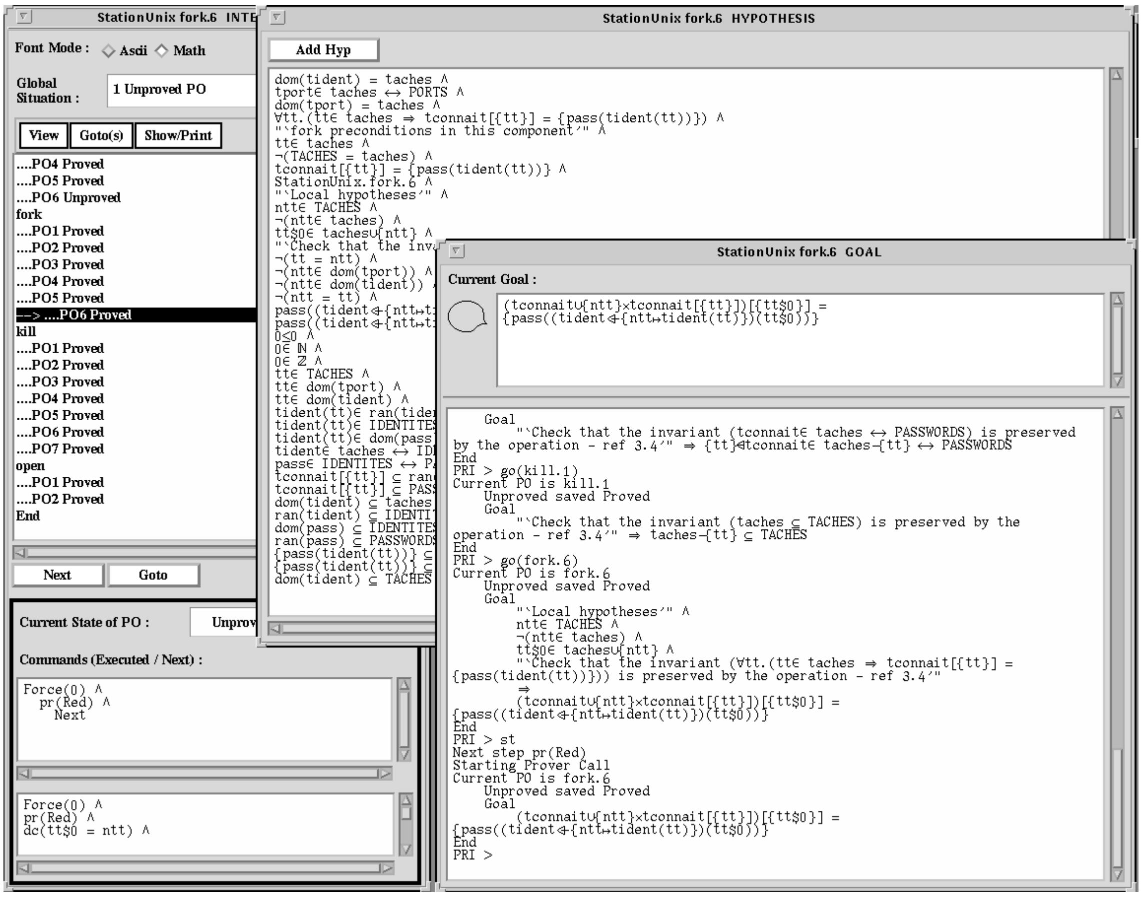

move mode


proof mode

In move mode, the global situation window only is used. The user checks the number undemonstrated POs meter, above the POs list. He/she can move within the POs using the buttons by the side of this list.

In proof mode, the user focuses mainly on the goal and command areas. It is in the command area that he/she performs his/her hypotheses (`sh`command), the hypotheses window being more specially reserved for a rapid and less controlled reading. After each proof command, the user must check the new proof status on the left of the command line area (refer section 6.4). During a proof, it often happens that reading a part of the component or of one of its adjoining levels becomes necessary. This is why it is advised to have iconized the corresponding files on the screen borders. When a consultation is needed, the user merely opens these windows that pop-up in front of the Prover one. After reading they must be iconized again to prevent any interference with the proof display.

These considerations on display might seem secondary. They in fact influence user concentration and notably have an impact on final efficiency. The method to organize display described here is not the sole one, the key point is a **rational display method**.



Global Situation : 1 Unproved PO

```
....PO4 Proved
....PO5 Proved
....PO6 Unproved
fork
....PO1 Proved
....PO2 Proved
....PO3 Proved
....PO4 Proved
....PO5 Proved
--> ....PO6 Proved
kill
....PO1 Proved
....PO2 Proved
....PO3 Proved
....PO4 Proved
....PO5 Proved
....PO6 Proved
....PO7 Proved
open
....PO1 Proved
....PO2 Proved
End
```

Current State of PO : Unprov

Commands (Executed / Next) :

```
Force(0) ∧
  pr(Red) ∧
    Next
```

```
Force(0) ∧
pr(Red) ∧
dc(tt$0 = ntt) ∧
```

```
dom(tident) = taches ∧
tporte∈ taches ↔ PORTS ∧
dom(tport) = taches ∧
∀tt.(tt∈ taches ⇒ tconnait[{tt}] = {pass(tident(tt))}) ∧
"`fork preconditions in this component´" ∧
tt∈ taches ∧
¬(TACHES = taches) ∧
tconnait[{tt}] = {pass(tident(tt))} ∧
StationUnix.fork.6 ∧
"`Local hypotheses´" ∧
ntt∈ TACHES ∧
¬(ntt∈ taches) ∧
tt$0∈ taches∪{ntt} ∧
"`Check that the inv
¬(tt = ntt) ∧
¬(ntt∈ dom(tport)) ∧
¬(ntt∈ dom(tident))
¬(ntt = tt) ∧
pass((tident⩤{ntt↦t
pass((tident⩤{ntt↦t
0≤0 ∧
0∈ ℕ ∧
0∈ ℤ ∧
tt∈ TACHES ∧
tt∈ dom(tport) ∧
tt∈ dom(tident) ∧
tident(tt)∈ ran(tide
tident(tt)∈ IDENTITES
tident(tt)∈ dom(pass
tident∈ taches ↔ ID
pass∈ IDENTITES ↔ P
tconnait[{tt}] ⊆ ran
tconnait[{tt}] ⊆ PASS
dom(tident) ⊆ taches
ran(tident) ⊆ IDENTI
dom(pass) ⊆ IDENTITES
ran(pass) ⊆ PASSWORDS
{pass(tident(tt))} ⊆
{pass(tident(tt))} ⊆
dom(tident) ⊆ TACHES
```

Current Goal :

```
(tconnait∪{ntt}×tconnait[{tt}])[{tt$0}] =
{pass((tident⩤{ntt↦tident(tt)})(tt$0))}
```

```
    Goal
        "`Check that the invariant (tconnait∈ taches ↔ PASSWORDS) is preserved
by the operation - ref 3.4´" ⇒ {tt}⩤tconnait∈ taches-{tt} ↔ PASSWORDS
End
PRI > go(kill.1)
Current PO is kill.1
    Unproved saved Proved
    Goal
        "`Check that the invariant (taches ⊆ TACHES) is preserved by the
operation - ref 3.4´" ⇒ taches-{tt} ⊆ TACHES
End
PRI > go(fork.6)
Current PO is fork.6
    Unproved saved Proved
    Goal
        "`Local hypotheses´" ∧
        ntt∈ TACHES ∧
        ¬(ntt∈ taches) ∧
        tt$0∈ taches∪{ntt} ∧
        "`Check that the invariant (∀tt.(tt∈ taches ⇒ tconnait[{tt}] =
{pass(tident(tt))})) is preserved by the operation - ref 3.4´"
        ⇒
        (tconnait∪{ntt}×tconnait[{tt}])[{tt$0}] =
{pass((tident⩤{ntt↦tident(tt)})(tt$0))}
End
PRI > st
Next step pr(Red)
Starting Prover Call
Current PO is fork.6
    Unproved saved Proved
    Goal
        (tconnait∪{ntt}×tconnait[{tt}])[{tt$0}] =
{pass((tident⩤{ntt↦tident(tt)})(tt$0))}
End
PRI >
```

Here we show a typical windows arrangement when the user is concentrating on a proof obligation: the command and goal window are in front, the command line area is visible on the left. The hypotheses window is inserted so that the last hypotheses are visible and near, by overflowing on the left while keeping a sufficient window size. By pressing a single key, the user can bring back the hypotheses to the front (we will not indicate *which* one as it depends on your environment).

6.3.2 The Interactive Prover in batch mode

As with all tools making up Atelier B, the Interactive Prover can be accessed in batch mode. Let us recall that batch mode is started with the `lanceBB` instead of `lanceAB`. The Automatic Prover is accessed with `pr(component,force)` and the Interactive Prover with `br`. Batch mode is described in Atelier B User Manual.

When batch mode (`br` command) is started, the proof is performed exactly as from the Motif interface command window - but the hypotheses and the command line are only in the Prover replying messages. The difficulties when using batch mode as compared with the Motif interface are:

- As there is no global situation window, you have to use the *GlobalSituation* (`gs` command) to get a list of all proof obligations.

- Lengthy information such as the hypotheses list scroll in the command window instead of being displayed in separate windows. We note for the hypotheses that when there are only additional hypotheses, the Prover does not return a full list but only the new hypotheses.
- The Interrupt button is not available in batch mode

But batch mode enables to use a non-graphic terminal and get a display in your standard command environment (such as a `xterm`) terminal).

We now give the record of a batch mode proof session corresponding to the demonstration used in section 6.1. To distinguish them, commands entered by the user are underlined. To start Atelier B in batch mode, use `lanceBB`.

```
PRVB% lanceBB

Beginning interpretation ...

bbatch 1> spl

Printing Projects list ...

      LIBRARY
      passwd

End of Projects list
```

We do not insist on Atelier B commands in batch mode - these are described in Atelier B User Manual. We now enter the *passwd* project :

```
bbatch 2> op passwd
bbatch 3> sml

Printing machines list ...

      StationUnix

End of machines list
```

We can now enter a *StationUnix* machine interactive proof session:

```
bbatch 4> b StationUnix
No current PO
```

We meet again the Interactive Prover command window display. A *GlobalSituation* provides us with the proof obligations list:

```
PRI> gs

State of all PO
```

```
    Initialisation
        PO1 Proved      {} <: TACHES
        PO2 Proved      {}: {} <-> PASSWORDS
        PO3 Proved      {}: {} +-> IDENTITES
        PO4 Proved      dom({}) = {}
        PO5 Proved      {}: {} <-> PORTS
    login
        PO1 Proved      tconnait\/(tport~[{pp}]\/{ntt})*{ww}: ...
        PO2 Proved      tident<+{ntt|->pass~(ww)}: taches\/{ntt}...
        PO3 Proved      dom(tident<+{ntt|->pass~(ww)}) = taches\/{ntt}
        PO4 Proved      tport\/{ntt|->pp}: taches\/{ntt} <-> PORTS
        PO5 Proved      dom(tport\/{ntt|->pp}) = taches\/{ntt}
        PO6 Unproved    (tconnait\/(tport~[{pp}]\/{ntt})*{ww})[{tt}] ...
    fork
        PO1 Proved      tconnait\/{ntt}*tconnait[{tt}]: taches\/{ntt}...
        PO2 Proved      tident<+{ntt|->tident(tt)}: taches\/{ntt} +->...
        PO3 Proved      dom(tident<+{ntt|->tident(tt)}) = taches\/{ntt}
        PO4 Proved      tport\/{ntt}*tport[{tt}]: taches\/{ntt} <-> PORTS
        PO5 Proved      dom(tport\/{ntt}*tport[{tt}]) = taches\/{ntt}
        PO6 Unproved    (tconnait\/{ntt}*tconnait[{tt}])[{tt$0}] = ...
    kill
        PO1 Proved      taches-{tt} <: TACHES
        PO2 Proved      {tt}<<|tconnait: taches-{tt} <-> PASSWORDS
        PO3 Proved      {tt}<<|tident: taches-{tt} +-> IDENTITES
        PO4 Proved      dom({tt}<<|tident) = taches-{tt}
        PO5 Proved      {tt}<<|tport: taches-{tt} <-> PORTS
        PO6 Proved      dom({tt}<<|tport) = taches-{tt}
        PO7 Proved      ({tt}<<|tconnait)[{tt$0}] = {pass(({tt}<<|...
    open
        PO1 Proved      tport\/{tt|->pp}: taches <-> PORTS
        PO2 Proved      dom(tport\/{tt|->pp}) = taches
End
No current PO
```

We decide to prove *fork* sixth obligation:

```
PRI> go(fork.6)
```

```
Current PO is fork.6
    Unproved saved Unproved
    Command line is
        Force(0) &
          Next
    Saved line pos 1
        Force(0) &
        dd
    Hypothesis
        "‘Component properties’" &
        pass: IDENTITES +-> PASSWORDS &
        pass~: PASSWORDS +-> IDENTITES &
        dom(pass) = IDENTITES &
        ran(pass) = PASSWORDS &
        TACHES: FIN(INTEGER) &
        not(TACHES = {}) &
        IDENTITES: FIN(INTEGER) &
        not(IDENTITES = {}) &
        PASSWORDS: FIN(INTEGER) &
        not(PASSWORDS = {}) &
        PORTS: FIN(INTEGER) &
        not(PORTS = {}) &
        btrue &
        "‘Component invariant’" &
        taches <: TACHES &
        tconnait: taches <-> PASSWORDS &
        tident: taches +-> IDENTITES &
        dom(tident) = taches &
        tport: taches <-> PORTS &
        dom(tport) = taches &
        !tt.(tt: taches => tconnait[{tt}] = {pa...
        "‘fork preconditions in this component’" &
        tt: taches &
        not(TACHES = taches) &
        tconnait[{tt}] = {pass(tident(tt))} &
        StationUnix.fork.6
    Goal
        "‘Local hypotheses’" &
        ntt: TACHES &
        not(ntt: taches) &
        tt$0: taches\/{ntt} &
        "‘Check that the invariant (!tt.(tt: ...
        =>
        (tconnait\/{ntt}*tconnait[{tt}])[{tt...
End
```

The proof obligation is displayed - with other status information (particularly the command line). We can try a *Prove* command:

```
PRI> pr

Starting Prover Call
Current PO is fork.6
    Unproved saved Unproved
    Command line is
        Force(0) &
          pr &
            Next
    Saved line pos 1
        Force(0) &
        dd
    New Hypothesis since last command
        "‘Local hypotheses’" &
        ntt: TACHES &
        not(ntt: taches) &
        tt$0: taches\/{ntt} &
        "‘Check that the invariant (!tt.(tt: taches...
        not(tt = ntt) &
        not(ntt: dom(tport)) &
        not(ntt: dom(tident)) &
        not(ntt = tt) &
        pass((tident<+{ntt|->tident(tt)})(tt$0)): ran(pass) &
        pass((tident<+{ntt|->tident(tt)})(tt$0)): PASSWORDS &
        0<=0 &
        0: NATURAL &
        0: INTEGER &
        tt: TACHES &
        tt: dom(tport) &
        tt: dom(tident) &
        tident(tt): ran(tident) &
        tident(tt): IDENTITES &
        tident(tt): dom(pass) &
        tident: taches <-> IDENTITES &
        pass: IDENTITES <-> PASSWORDS &
        tconnait[{tt}] <: ran(tconnait) &
        tconnait[{tt}] <: PASSWORDS &
        dom(tident) <: taches &
        ran(tident) <: IDENTITES &
        dom(pass) <: IDENTITES &
        ran(pass) <: PASSWORDS &
        {pass(tident(tt))} <: ran(tconnait) &
        {pass(tident(tt))} <: PASSWORDS &
        dom(tident) <: TACHES &
        ntt|->tident(tt): TACHES*dom(pass) &
        ntt|->tident(tt): TACHES*IDENTITES &
        ntt|->tident(tt): TACHES*ran(tident) &
        pass(tident(tt)): ran(pass) &
        pass(tident(tt)): PASSWORDS &
        ntt = tt$0 &
        not(tt = tt$0) &
        not(tt$0 = tt) &
        tconnait[{tt$0}] <: ran(tconnait) &
        tconnait[{tt$0}] <: PASSWORDS
    Goal
        tconnait[{tt$0}]\/{pass(tident(tt))} =
        {pass((tident<+{ntt|->tident(tt)})(tt$0))}
End
```

Note all the new derived hypotheses created by the Automatic Prover. In batch mode, only the new hypotheses are displayed at each step unless we move back in the proof tree (for instance, with the *Back* command), as we will see in the next step. We move back to the `pr` command, according to the general method:

```
PRI> ba

Current PO is fork.6
    Unproved saved Unproved
    Command line is
        Force(0) &
          Next
    Saved line pos 1
        Force(0) &
        dd
    Hypothesis
        "‘Component properties’" &
        pass: IDENTITES +-> PASSWORDS &
        ...
        tconnait[{tt}] = {pass(tident(tt))} &
        StationUnix.fork.6
    Goal
        "‘Local hypotheses’" &
        ntt: TACHES &
        not(ntt: taches) &
        tt$0: taches\/{ntt} &
        "‘Check that the invariant (!tt.(tt: ...
        =>
        (tconnait\/{ntt}*tconnait[{tt}])[{tt$0}] =
        {pass((tident<+{ntt|->tident(tt)})(tt$0))}
End
```

To shorten this text, all hypotheses are not given. Note the command line evolution under the “`command line is ...`” clause. We are back to the beginning of the demonstration; we can try and apply the Predicate Prover:

```
PRI> dd & pp

Starting Deduction
Starting Prover Predicate Call
EXECUTION ABORTED ON GOAL: long, dumping in /tmp/lastfrm...

StationUnix.but interrupted
The Predicate Prover went out of time
Current PO is fork.6
    Unproved saved Unproved
    Command line is
        Force(0) &
          dd &
```

```
            Next
    Saved line pos 2
        Force(0) &
        dd
    New Hypothesis since last command
        ...
    Goal
        (tconnait\/{ntt}*tconnait[{tt}])[{tt$0}] =
        {pass((tident<+{ntt|->tident(tt)})(tt$0))}
End
```

We will now repeat the demonstration given in section 6.1 quickly. We move to the beginning using a *Reset* command:

```
PRI> re

Resetting PO
Current PO is fork.6
    Unproved saved Unproved
    Command line is
        Force(0) &
          Next
    Saved line pos 1
        Force(0) &
        dd
    Hypothesis
        ...
    Goal
        "‘Local hypotheses’" &
        ntt: TACHES &
        not(ntt: taches) &
        tt$0: taches\/{ntt} &
        "‘Check that the invariant (!tt.(tt...
        =>
        (tconnait\/{ntt}*tconnait[{tt}])[{tt$0}] =
        {pass((tident<+{ntt|->tident(tt)})(tt$0))}
End
```

The demonstration previously seen can start:

```
PRI> pr(Red)

Starting Prover Call
Current PO is fork.6
    Unproved saved Unproved
    Command line is
        Force(0) &
          pr(Red) &
            Next
    Saved line pos 1
        Force(0) &
        dd
    New Hypothesis since last command
        ...
    Goal
        (tconnait\/{ntt}*tconnait[{tt}])[{tt$0}] =
        {pass((tident<+{ntt|->tident(tt)})(tt$0))}
End

PRI> dc(ntt = tt$0)

Starting Do Cases
Current PO is fork.6
    Unproved saved Unproved
    Command line is
        Force(0) &
          pr(Red) &
            dc(ntt = tt$0) &
              Next
    Saved line pos 1
        Force(0) &
        dd
    New Hypothesis since last command

    Goal
        ntt = tt$0 =>
          (tconnait\/{ntt}*tconnait[{tt}])[{tt$0}] =
          {pass((tident<+{ntt|->tident(tt)})(tt$0))}
End

PRI> pr(Red)

Starting Prover Call
Current PO is fork.6
    Unproved saved Unproved
    Command line is
        Force(0) &
          pr(Red) &
            dc(ntt = tt$0) &
              pr(Red) &
                Next
    Saved line pos 1
        Force(0) &
        dd
    New Hypothesis since last command
        ntt = tt$0 &
        not(tt = tt$0) &
        not(tt$0 = tt) &
        pass(tident(tt)): ran(pass) &
        pass(tident(tt)): PASSWORDS
    Goal
        (tconnait\/{tt$0}*tconnait[{tt}])[{tt$0}] =
        {pass(tident(tt))}
End

PRI> pp

Starting Prover Predicate Call
Proved by the Predicate Prover
Current PO is fork.6
    Unproved saved Unproved
    Command line is
        Force(0) &
          pr(Red) &
            dc(ntt = tt$0) &
              pr(Red) &
                pp &
              Next
    Saved line pos 1
        Force(0) &
        dd
    Hypothesis
        ...
    Goal
        not(ntt = tt$0) =>
         (tconnait\/{ntt}*tconnait[{tt}])[{tt$0}] =
         {pass((tident<+{ntt|->tident(tt)})(tt$0))}
End

PRI> pr(Red)

Starting Prover Call
Current PO is fork.6
    Unproved saved Unproved
    Command line is
        Force(0) &
          pr(Red) &
            dc(ntt = tt$0) &
              pr(Red) &
                pp &
              pr(Red) &
                Next
    Saved line pos 1
        Force(0) &
        dd
    New Hypothesis since last command
        ...
    Goal
        (tconnait\/{ntt}*tconnait[{tt}])[{tt$0}] =
        {pass(tident(tt$0))}
End

PRI> pp

Starting Prover Predicate Call
Proved by the Predicate Prover
Current PO is fork.6
    Proved saved Unproved
    Command line is
        Force(0) &
          pr(Red) &
            dc(ntt = tt$0) &
              pr(Red) &
                pp &
              pr(Red) &
                pp &
          Next
    Saved line pos 1
        Force(0) &
        dd
    Hypothesis
        ...
    Goal
        "‘Local hypotheses’" &
        ntt: TACHES &
        not(ntt: taches) &
        tt$0: taches\/{ntt} &
        "‘Check that the invariant (!tt.(tt...
        =>
        (tconnait\/{ntt}*tconnait[{tt}])[{tt$0}] =
        {pass((tident<+{ntt|->tident(tt)})(tt$0))}
End
```

We can quit the Interactive Prover and Atelier B. We note that the `h` command in the interactive mode displays a list of all available commands:

```
PRI> qu

PO fork.6 saved
bbatch 6> h


General commands :
(cd  ) change_directory
(ddm ) disable_dependence_mode
(erf ) edit_res_file
```

```
(eur ) edit_users_res
(edm ) enable_dependence_mode
(h   ) help
(hh  ) html_help
(hph ) html_prover_help
(hrb ) html_rules_base
...

Project level commands :
(add ) add_definitions_directory
(apl ) add_project_lib
(apr ) add_project_reader
...

Machine level commands (available after open_project) :
(aa  ) ada_all               (a   ) adatrans
(af  ) add_file              (ani ) animator
(b0c ) b0check               (b   ) browse
...

bbatch 7> clp
bbatch 8> q

End of interpretation (8 lines)
PRVB%
```

6.4 The command line

To guide the user during his/her demonstration, the Prover provides information displaying the proof tree as an indented list: the command line. This information is displayed in the command line area in the global situation window. Command indentation in this area is crucial, it represents the various proof paths. We are to explain its use with examples.

Let us assume that the command line display is as follows:

```
Force(0) &
  dc(var = TRUE) &
    pr &
    pr &
  Next
```

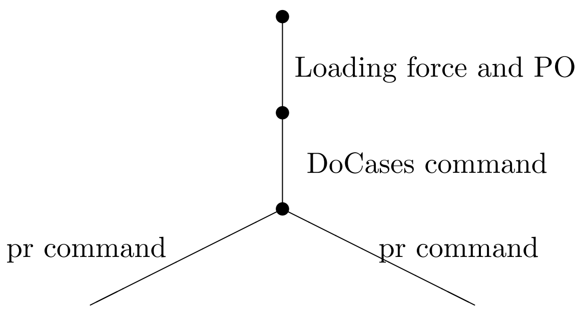
This means that the proof process has been:

- Prover set in force 0;
- in this force, the user has accessed a PO. Such access is not represented but under

the action of force 0, a goal has been created.

- on this initial goal, the user has entered a *DoCases* (`dc` command to create two cases: either $var = \mathsf{TRUE}$ or $\neg(var = \mathsf{TRUE})$.
- *DoCases* (`dc`) several son goals; the user has called the Automatic Prover to process the first son goal.
- the user has called again the Automatic Prover to process the second son goal.
- the `Next` keyword is offset only in relation with `Force(0)`, which means that the next goal to process is not a son of the PO initial goal. That is, this PO proof is completed - which is indicated by the green color of the goal area.

In this example, the demonstration is successful. The interface indicates it by coloring in sharp green the goal area and prohibiting other proof commands. In our example, the proof tree is:



As the reader will have understood, an interactive demonstration is aimed at bringing back to the left the `Next` keyword. This keyword position enables to know whether the current goal is a son of the previous goal. A typical case where the localization with the command line is required is the one where a `pr(Red)` command has created several sub-goals. For example:

```
Force(0) &
  pr(Red) &
    ah(not(EE = {})) &
      pr &
      pr &
    Next
```

Here, the user has first launched the Automatic Prover in reduced mode. It fails on one of the sub-goals and the user helps it by adding a hypothesis (*AddHypothesis* command), that he/she demonstrates by `pr`, a second `pr` enabling then to demonstrate the blocking sub-goal. A new goal appears that does not appear to have any relation with the previous one: this is normal as it is the next sub-goal created by the initial `pr(Red)` command. But when the user is not used to following the demonstration in the command line, such a process can be surprising.

In our example, the call to the Automatic Prover was terminated when the first sub-goal failed. Maybe we can complete the demonstration by starting it again?

```
Force(0) &
  pr(Red) &
    ah(not(EE = {})) &
      pr &
      pr &
    pr &
  Next
```

6.5 Simple proof strategies

Among all various strategies used case by case by the user for interactive demonstrations, there are groups of commands that often return together. The joint use of these commands appears to be kinds of *elementary strategies* from which complex demonstrations are built.

We have identified four basic strategies:

1. *Prove* and *PredicateProver* ;
2. *AddHypothesis* and *DoCases* ;
3. *SearchRule* and *ApplyRule* ;
4. user rules and *ApplyRule*.

Such a division is indeed rather subjective. We shall nonetheless present these four strategies.

6.5.1 Prover and Predicate Prover

A demonstration can often be conducted simply by alternating calls to the Automatic Prover and to the Predicate Prover. Normally, the general method presented in this chapter should make such a method implicit, as it always stipulates trying the `pr` and `pp` commands before any other one.

The `pp` command takes 60 seconds before failing when it cannot prove the current goal - this might be rather too long to systematically try it on each new goal. This is why it frequently happens that one notices afterwards that only one sub-goal could be demonstrated by `pp` - when another more complex demonstration has been attempted. The reader will refer section 6.2.4 that give elements to foresee when 6.2.4 has chances to succeed as well as how to start 6.2.4 with shorter delays or on reduced hypotheses.

6.5.2 Adding hypotheses and proof by cases

The (*AddHypothesis*, `ah`commands) and the proof by cases (*DoCases*, `dc`command) enable a key action on the demonstration as they give the user the possibility to *introduce new expressions.*

Adding hypotheses: an example

We have to demonstrate the following lemma:

$$\begin{array}{l} c1 \in 0 \,..\, 100 \;\wedge \\ c2 \in 0 \,..\, c1 \;\wedge \\ c3 \in 0 \,..\, c2 \;\wedge \\ c4 \in 0 \,..\, c3 \\ \;\Rightarrow \\ c1 + c2 + c3 + c4 \in 0 \,..\, 400 \end{array}$$

The Automatic Prover in force 0 and the Predicate Prover fail on this lemma. According to the general method, we start by calling the Automatic Prover in reduced mode:

```
PRI > pr(Red)

Starting Prover Call
Current PO is Initialisation.1
    Unproved saved Unproved
    Goal
        0<=400-c1-c2-c3-c4
End
```

We have now to get variation intervals independent from other variables for *c2* to *c4*, that is we have to 'unconnect' the variables. We suggest to the Prover to perform this operation, starting by *c2* that depends only from *c1* :

```
PRI > ah(c2<=100)

Starting Add Hypothesis
Current PO is Initialisation.1
    Unproved saved Unproved
    Goal
        c2<=100
End
```

The Prover now tries to demonstrate this first simple result. We start the Automatic Prover again:

```
PRI > pr

Starting Prover Call
Current PO is Initialisation.1
    Unproved saved Unproved
    Goal
```

```
        c2<=100 => 0<=400-c1-c2-c3-c4
End
```

This intermediate result has been directly proven, which can be seen from the reappearance of the initial goal under this form: $H \Rightarrow B$ where H is the added hypothesis. The `pr` command does not yet complete the lemma (the test is not represented). We can load this hypothesis with a simple deduction command:

```
PRI > dd

Starting Deduction
Current PO is Initialisation.1
    Unproved saved Unproved
    Goal
        0<=400-c1-c2-c3-c4
End
```

We proceed similarly with *c3*:

```
PRI > ah(c3<=100)

Starting Add Hypothesis
Current PO is Initialisation.1
    Unproved saved Unproved
    Goal
        c3<=100
End

PRI > pr

Starting Prover Call
Current PO is Initialisation.1
    Unproved saved Unproved
    Goal
        c3<=100 => 0<=400-c1-c2-c3-c4
End

PRI > pr

Starting Prover Call
Current PO is Initialisation.1
    Proved saved Unproved
    Goal
        "‘Check ...’" => c1+c2+c3+c4: 0..400
End
```

After adding the hypothesis on *c3*, the Automatic Prover was able to complete the demonstration alone as we see in the last `pr` command (the new status is `Proved saved Unproved`, which means that current status is 'Proved' and that we have not yet saved. The final proof tree is:

```
  Force(0) &
    pr(Red) &
      ah(c2<=100) &
        pr &
        dd &
          ah(c3<=100) &
            pr &
            pr &
  Next
```

In this demonstration, we have guided the Prover by proposing intermediate expressions to be demonstrated. We could have this demonstration succeed without searching for a rule.

An example of proof by cases

Let us demonstrate the following lemma:

> "'*Local hypotheses*'" $\wedge$
> $xx \in \{7, 6, 5, 4, 3, 2, 1\}$ $\wedge$
> "'*Check ... ref 3.3*'"
> $\Rightarrow$
> $xx \in \{1, 2, 3, 4, 5, 6, 7\}$

The Automatic Prover in force 0 and the Predicate Prover fail on this lemma. In fact, the only way to prove this is to check the equality of each of the $\{7, 6, 5, 4, 3, 2, 1\}$ and $\{1, 2, 3, 4, 5, 6, 7\}$ sets. This amounts to making seven cases on xx value. Such proof by cases are seldom automatically triggered, proofs by multiple cases requiring long delays. We start the proof by cases with a *DoCases* command, preceeded by a reduced call to the Automatic Prover - according to the general method.

PRI > pr(Red)

```
Starting Prover Call
Current PO is Initialisation.1
    Unproved saved Unproved
    Goal
        xx = 1
End
```

It is useless to check where the new created goal comes from; we clearly have to start a proof by cases:

PRI > dc(xx, {7,6,5,4,3,2,1})

```
Do Cases on Enumerated {7}\/{6}\/{5}\/{4}\/{3}\/{2}\/{1}
Current PO is Initialisation.1
    Unproved saved Unproved
    Goal
        xx = 7 => xx = 1
```

```
End
```

We have used `dc` (refer Prover Reference Manual, [?]), that is `dc(v,E)` where `v` is a variable belonging to the `E` numerable set. The Prover must then demonstrate $v \in E$ - here a direct hypothesis - then goes into each $v = e_i$ case for each e_i element of E. In our example, each case is directly demonstrated:

```
PRI > pr

Starting Prover Call
Current PO is Initialisation.1
    Unproved saved Unproved
    Goal
        xx = 6 => xx = 1
End

PRI > pr

Starting Prover Call
Current PO is Initialisation.1
    Unproved saved Unproved
    Goal
        xx = 5 => xx = 1
End

PRI > pr

Starting Prover Call
Current PO is Initialisation.1
    Unproved saved Unproved
    Goal
        xx = 4 => xx = 1
End

PRI > pr

Starting Prover Call
Current PO is Initialisation.1
    Unproved saved Unproved
    Goal
        xx = 3 => xx = 1
End

PRI > pr

Starting Prover Call
Current PO is Initialisation.1
    Unproved saved Unproved
    Goal
        xx = 2 => xx = 1
End
```

```
PRI > pr

Starting Prover Call
Current PO is Initialisation.1
    Unproved saved Unproved
    Goal
        xx = 1 => xx = 1
End

PRI > pr

Starting Prover Call
Current PO is Initialisation.1
    Proved saved Unproved
    Goal
        "‘Local hypotheses’" &
        xx: {7,6,5,4,3,2,1} &
        "‘Check ... ref 3.3’"
        =>
        xx: {1,2,3,4,5,6,7}
End
```

In this example, we have used a single proof command: *DoCases* to start a proof with more cases than authorized in automatic proof. The other commands are only calls to the Automatic Prover. Proof by cases is often useful as soon as the lemma contains numerated or numerable sets or specific cases of variables values.

6.5.3 Searching and applying database rules

The Prover rule database can be used in interactive mode; this prevents from adding rules whose validity is not guaranteed and will have to be demonstrated later. When it seems that the current goal can be simplified by using a simple mathematical rule, we search for the corresponding rule with *SearchRule* and apply it with *ApplyRule*. We have here an example of this method:

Let us consider the following lemma:

$$\begin{array}{l} \mathit{ff} \in \mathbb{N} \;+\!\!\!\!\rightarrow\; \mathbb{N} \wedge \\ \mathit{ff}[0\,..\,50] \subseteq 0\,..\,100 \wedge \\ \mathit{ff}[51\,..\,100] \subseteq 0\,..\,100 \\ \;\Rightarrow \\ \mathit{ff}[0\,..\,50 \cup 51\,..\,100] \subseteq 0\,..\,100 \end{array}$$

Let us recall that $f[E]$, the functional image, designates by f the set of images of E elements. The functional image is equal to the union of each subset functional image. Although this rule seems clear it would have to be demonstrated exactly to use it in a demonstration. Luckily, as we are going to see this rule is present in the rules database.

This rule would enable to rewrite the goal as:

$\mathit{ff}[0 \,..\, 50] \cup \mathit{ff}[51 \,..\, 100] \subseteq 0 \,..\, 100$
which is more directly to the hypotheses. We have the *intuition* that this simpler goal would be demonstrated. To verify this, we can add this expression by an `ah` command:

```
    ah(ff[0..50]\/ff[51..100] <: 0..100)
```

A simple `pr` command demonstrate this hypothesis. We have used *ddHypothesis* to test if a goal is easily demonstrated. We can return to the beginning of the obligation and search for the adequate rule: :

```
PRI > sr(Rewr,(f[a \/ b]))
```

Let us analyze the syntax of the *SearchRule* command: the `Rewr` keyword indicates that we are looking for a rewriting rule in order to transform $\mathit{ff}[0 \,..\, 50 \cup 51 \,..\, 100]$ that is a *sub-form* of the goal. The $f[a \cup b]$ filter that the left term in the rewriting must contain this formula. The Prover answers:

```
Searching in Rewr rules with filter
    consequent should contain f[a\/b]
Starting search...
Rule list is
    SimplifyRelImaLongXY.6       (Backward)
        r[u\/v] == r[u]\/r[v]
End of rule list
```

The rule does exist. We note that jokers used, r, u et v are not those used to search: joker names are of no consequence. In fact, a rule with the `r[u\/(f(x))] == ...` form would have been equally used. We can apply this rule:

```
PRI > ar(SimplifyRelImaLongXY.6, Goal)

Starting Apply Rule
Current PO is Initialisation.1
    Unproved saved Unproved
    Goal
        ff[0..50]\/ff[51..100] <: 0..100
End
```

We complete the demonstration:

```
PRI > pr

Starting Prover Call
Current PO is Initialisation.1
    Proved saved Unproved
    Goal
        "‘Check ... 3.3’" => ff[0..50\/51..100] <: 0..100
End
```

In this example, we have seen how to search for a rule we had an idea of, instead of writing it in a .pmm file. We can thus be sure that this rule is valid.

The rule we have used belongs to a theory with a name containing the word `Long`. Theories like this lengthen the expressions they transform. These rules are never used by the Automatic Prover as they represent additional loop risks. They are thus reserved solely for an interactive use.

6.5.4 User rules

The last basic proof technique is adding user rules. This must be avoided when possible as the added rules will have to be proven later.

The method to add and use a user rule is described in section 6.2.3, with an application example. The reader will refer to this section.

6.6 Advanced use

We give here various elements enabling a better use of the Automatic Prover:

- Final proof checking;
- Using an admission rule;
- Optimizing moves in the proof;
- Selecting a higher force in the proof;
- Tracing a demonstration.

6.6.1 Final proof checking

In formal proof phase, the user works on each proof obligation one after the other and at times for a lengthy period. Each successful proof obligation is labeled 'Proved' in the component management file and its proof is not replayed. Although not replaying proofs already performed is crucial to allow for good productivity, there are situations where proofs cannot be replayed.

For example, a demonstration can succeed by using a rule; when the user modifies this rule during the demonstration of *another* proof obligation, it could be that this demonstration does not succeed. As it is not replayed, the PO status is still 'Proved'. Moreover, during the proof the user can have added 'temporary' rules he/she is not sure about. When these rules are deleted, we must check that the demonstrations can nonetheless be replayed. Such proof replaying is done at the end, or more seldom from time to time on components whose interactive proof takes several days.

For such replays, the recommended method is as follows:

1. In Atelier B, at project level (outside the Interactive Prover), select the concerned component and the *Unprove* option in the *Prove* menu. This resets all obligations to unproved status.

2. In the same *Prove* menu, select the *Automatic (Replay)* option. This triggers the replay of all saved demonstrations, that is: a PO demonstrated in force 0 is replayed in force 0, a PO demonstrated in force 2 is replayed in force 2, a PO demonstrated in force 1 with interactive commands is replayed in force 1 with these command, etc.

3. Check that you have effectively recovered the initial proof status, with the same number of proven obligations.

This operation must equally be performed to check a project proof when received. Caution: this can be very lengthy; fully replaying several thousand proof obligations could require a lot of CPU time.

6.6.2 Using an admission rule

An admission rule is a user rule enabling to prove anything. You simply add in the `<component>.pmm` user rules file :

```
THEORY Admis IS
  bcall(WRITE: bwritef(‘‘A’’)) => p
END
```

The consequence of this rule is the `p`joker that coincides with any goal. Such a rule is patently false. It must be removed and demonstrations be replayed to ensure that none of them use it: refer section 6.6.1.

The `bcall(...)` antecedent is a theory language directive that enables to write `A` as 'admitted' juste before the `+` that indicates success in automatic proof - which enables to betray the use of these rules when replaying a proof. We can enter on the terminal the accepted goal:

```
bcall(WRITE: bwritef(‘‘admis : %\n’’,p)) => p
```

Using an admission rule is strongly recommended to conduct the proof efficiently. The admission rule is accessed with *ApplyRule* : `ar(Admis.1,Once)`. It has two main uses:

- Case control: the Automatic Prover decides at times to perform a proof by cases - not always judiciously. The admission rule enables to know these cases. For example:

```
pr &
  ar(Admis.1,Once) &
  ar(Admis.1,Once) &
  ar(Admis.1,Once) &
Next
```

The first call to the Automatic Prover has created three cases. Refer section 6.8.1.

- Validating the usefulness of an added hypothesis: the *AddHypothesis* command enables to add a demonstrable hypothesis from existing ones. This demonstration can be complex; it is then better to be sure this addition is useful before performing it. For example:

```
ah(not(xx = e1)) &
  ar(Admis.1,Once) &
  pr &
Next
```

  The $xx \neq e1$ hypothesis usefulness being established, we can return and demonstrate it.

Using the admission rule obliges us to go back to the demonstration - refer section 6.6.3: some precautions are to be taken. The admission rule is at times used in a tuning-up phase - refer to section 5.4.6.

For reasons of division of work, it may be useful to create reduced admission rules that can act on specific proof obligation only. For example:

```
THEORY Admis IS

  binhyp(C.operation1.n) &
  bcall(WRITE: bwritef(``A''))
  =>
  p;

  binhyp(C.operation2.3) &
  bcall(WRITE: bwritef(``A''))
  =>
  p

END
```

These rules are based on the presence of a non-mathematical hypothesis of the form: `component.operation.number` systematically added by the Prover when loading a proof obligation. This hypothesis, called a spotting hypothesis is besides displayed in the interface.

The admission theory above accepts all POs from operation1 and PO 3 from operation2. The theory language binhyp directive allows the operation of a rule only when the argument formula is in the hypotheses. This theory can be used without having to quote the rule numbers thanks to the `ar(Admis)` command - this results in the application of all rules from the `ar(Admis)` command as long as one of these apply.

Beware of offset numbers: it is important to place the admission rules in a separate theory. Thus, it will be possible to remove them at the end without having the other rules lose their number - which would destroy the user demonstrations using these rules. Indeed, use of a rule with an *ApplyRule* command is done by *specifying the rule by its order in the membership theory).* Deleting rules in a theory might offset these numbers (refer section 6.8.2).

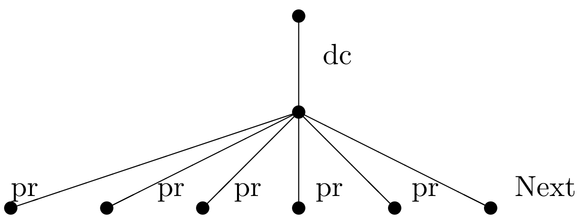
6.6.3 Moving in a proof

The proof control system enable back moves. It is thus possible to perform very long demonstrations; at each new command the Prover proceeds from the previous status without replaying everything - and in case of an error, back moves are sufficient. This proof method is based on the *Logic Solver* native functionalities.

The underlying *Logic Solver* proof management system influence moves performance. Back moves on completed proof paths are notably longer. For example:

```
dc(xx,{e1,e2,e3,e4,e5,e6}) &
  pr &
  pr &
  pr &
  pr &
  pr &
  Next
```

This corresponds to the proof tree:



When the user activates the *Back* (`ba`) command, the system must find out the proof by the cases previous goal which has been unloaded with the `pr` command. The *Logic Solver* keeps all current goals in the current path only. Keeping the paths of all completed proofs would require too much memory. The only way to find out this goal is to return to the *DoCases* (excluded) then to delete intermediate goals by calling repeatedly to the Automatic Prover. There are thus 4 calls to the Automatic Prover to be made. You can see the problem when each call take one minute! In this specific example, it would be better to delete all intermediate goals by admission, instead of replaying their demonstration. But you must understand that such an optimization is not a general method.

In another way, a *Back* command can trigger the replay of full parts of a demonstration to re-create a disappeared goal. Besides these replays are pointed to by messages stating the beginning of each replayed command: `Starting Prover Call...` etc. But the *Back* command is immediate when there is nothing to re-create, that is if it returns to a goal whose son is the current goal. In the previous example, we immediately return just before the *DoCases* command.

The `ba(Node)` command enables to return to the nearest parent goal; it is always immediate as it doesn't start a command. In the previous example, `ba(Node)` enables to return directly before the *DoCases* command. In fact, a command always returns to the previous indent level in the command line. For example, if the status before the command is as follows:

```
ah(not(xx = e1)) &
  dc(zz,ENUM) &
    pr &
    pr &
      ah(ww = 5) &
        pr &
        pr &
    Next
```

To find the new status after a `ba(Node)` command, we have only to place `Next` instead of the nearest directly inferior indent command, going upwards:

```
ah(not(xx = e1)) &
  Next
```

In this example, the user could have entered five times `ba`, or do `ba(5)` to get the same result. This would be a very bad method, much more lengthy.

6.6.4 Choosing a higher force

Choosing a higher force to attempt a proof according to a PO form is always very speculative. Indeed it is not possible to foresee exactly what the Automatic Prover will do in this or that force. To do so one would have to know all the rules and mechanisms. Such attempts are thus guided by intuition and experience.

The higher forces are force 1, 2 and 3. Each contains all mechanisms from the previous force - this is not the case with force 0 whose mechanisms are not systematically included in force 1. Higher forces have also increasing dimension constants: force 2 may create more sub-cases than force 1, derive more hypotheses, etc.

As a guide in selecting a higher force, it is better to only remember the specificities of each force, which are:

- **Force 1**: hypotheses are processed by entering the Prover. They are thus more split up than in force 0. For example, any hypothesis of the form: $x \in a\,..\,b$ will be systematically transformed into $a \leq x$ et $x \leq b$.
- **Force 2**: specific hypotheses are created according to expressions found in the goal. For example, when the goal contains $a \times b$ produce with $0 \leq a$ and when there is a c variable such as $c \leq b$, then the $a \times c \leq a \times b$ hypothesis is added.
- **Force 3**: attempt rules are used. For example, when the goal is to prove $a \leq b$ and if $a \leq c$ is a hypothesis, then a sub-proof will be started to try and demonstrate $c \leq b$.

6.6.5 Proof tracing

We call proof tracing the information produced during a demonstration and enabling to know how it was performed. Tracing is notably useful when using automatic provers, as it provides a justification of the demonstration.

There are two kinds of proof tracing designed within Atelier B: the production of a written mathematical demonstration or the step by step tracing of a call to the Automatic Prover. This latter is used to understand new goals that might appear after a call to the Automatic Prover. We shall examine the tracing to the Automatic Prover on an example.

Before this example, let us insist on the fact that *tracing is seldom used to make a proof successful.* Indeed, it is mostly useless to understand what made the last call to the Automatic Prover, what is crucial is to see how to proceed to the next goal. Most of times, a tracing produced during an unsuccessful demonstration is a loss of time. This is why we have chosen in the following example an obligation tracing that is demonstrated by a single call to the Automatic Prover in force 1: proof tracing being thus displayed in its true after the event information function.

We have to demonstrate the following lemma:

$$
\begin{array}{l}
PORTS \in \mathbb{F}(\mathbb{Z}) \land \\
PORTS \subseteq \mathbb{Z} \land \\
\neg(PORTS = \varnothing) \land \\
taches \subseteq TACHES \land \\
tport \in taches \leftrightarrow PORTS \land \\
\mathsf{dom}(tport) \subseteq taches \land \\
\mathsf{ran}(tport) \subseteq PORTS \land \\
\mathsf{dom}(tport) \subseteq TACHES \land \\
\mathsf{dom}(tport) = taches \land
\end{array}
$$

$$\begin{array}{l} tt \in taches \wedge \\ tt \in \mathsf{dom}(tport) \wedge \\ tt \in TACHES \wedge \\ \neg(TACHES = taches) \\ \ \Rightarrow \\ \{tt\} \lhd tport \in taches - \{tt\} \ \leftrightarrow \ PORTS \end{array}$$

To get a tracing of the demonstration, simply start the Interactive Prover, access this PO (when then are in force 1) and enter:

```
    pr(Ru.Goal.None, File)
```

It is a `pr` c with arguments to specify that we want a proof tracing stating goals and rules, displayed on the terminal and saved. The tracing display is as follows:

```
  Starting Trace in mode Ru.Goal.None , File

Starting Prover Call
  After deduction, goal is now
        {tt}<<|tport: taches-{tt} <-> PORTS
```

The deduction enables to load local hypotheses.

```
  Attempt to prove
        not(tt: taches)
```

The goal containing expressions that can be simplified if *tt* is not in *taches*, the Prover tries to demonstrate this proposition

```
  Obvious goal tt: taches  is discharged because in hypothesis.
  HiddenPredicate mechanism is transforming goal
       not(tt: taches)
  in
       bfalse
  Obvious goal StationUnix.kill.5  is discharged because in hypothesis.
  Obvious goal "‘Check ...4’"  is discharged because in hypothesis.
  After deduction, goal is now
        bfalse
  Attempt to prove
        bfalse
  fails.
```

$\neg(tt \in taches)$ demonstration fails. The $tt \in taches$ predicate is indeed is a hypothesis; it is replaced with **btrue** in the hypothesis to be proven. It is the *HiddenPredicate* mechanism that performs this replacement. Before concluding to a failure, the last two hypotheses: `StationUnix.kill.5` and `"‘Check ...4’"` have been transmitted to the Prover to attempt a simplification.

To sum up: the Prover made a complete cycle to remark that the $\neg(tt \in taches)$ proposition is false and that it is not thus possible to simplify the goal by using it. In such a proof tracing, every part of the Prover that has been tried is traced. Moreover, the mechanism names are given; unless you are a Prover mechanisms specialist, you cannot know all these names. We thus advise you not to spend time on each step of the tracing; the key is to *globally* understand what has happened. Let us proceed with our reading:

```
By applying atomic rule InRelationXY.1,
     dom(a): POW(s) &
     ran(a): POW(t)
     =>
     a: s <-> t
the goal {tt}<<|tport: taches-{tt} <-> PORTS  is now
     dom({tt}<<|tport) <: taches-{tt}
 and ran({tt}<<|tport) <: PORTS
```

A rule has been applied; we now have the two above goals.

```
Attempt to prove
      not(tt: taches)
Obvious goal tt: taches  is discharged because in hypothesis.
HiddenPredicate mechanism is transforming goal
     not(tt: taches)
in
     bfalse
Obvious goal "‘Check ...4’"  is unloaded because in hypothesis.
After deduction, goal is now
      bfalse
Attempt to prove
      bfalse
fails.
```

The Prover has again tried to prove $\neg(tt \in taches)$.

```
Goal
     dom({tt}<<|tport) <: taches-{tt}
is simplified in
     dom(tport)-{tt} <: taches-{tt}
```

The goal has been simplified by a rewriting rule. In the tracing mode we selected, these rewriting rules are not traced: we can get additional tracing by using the `pr(Ru.Goal.None, File, Simpl)` command. Caution, this induces lengthy tracing.

```
By applying atomic rule InPOWXY.24,
     a: POW(c\/b)
     =>
     a-b: POW(c)
the goal dom(tport)-{tt} <: taches-{tt}  is now
     dom(tport) <: taches-{tt}\/{tt}

Goal
     dom(tport) <: taches-{tt}\/{tt}
```

```
  is simplified in
       dom(tport) <: taches
  Obvious goal dom(tport) <: taches  is discharged because in
       hypothesis.
```

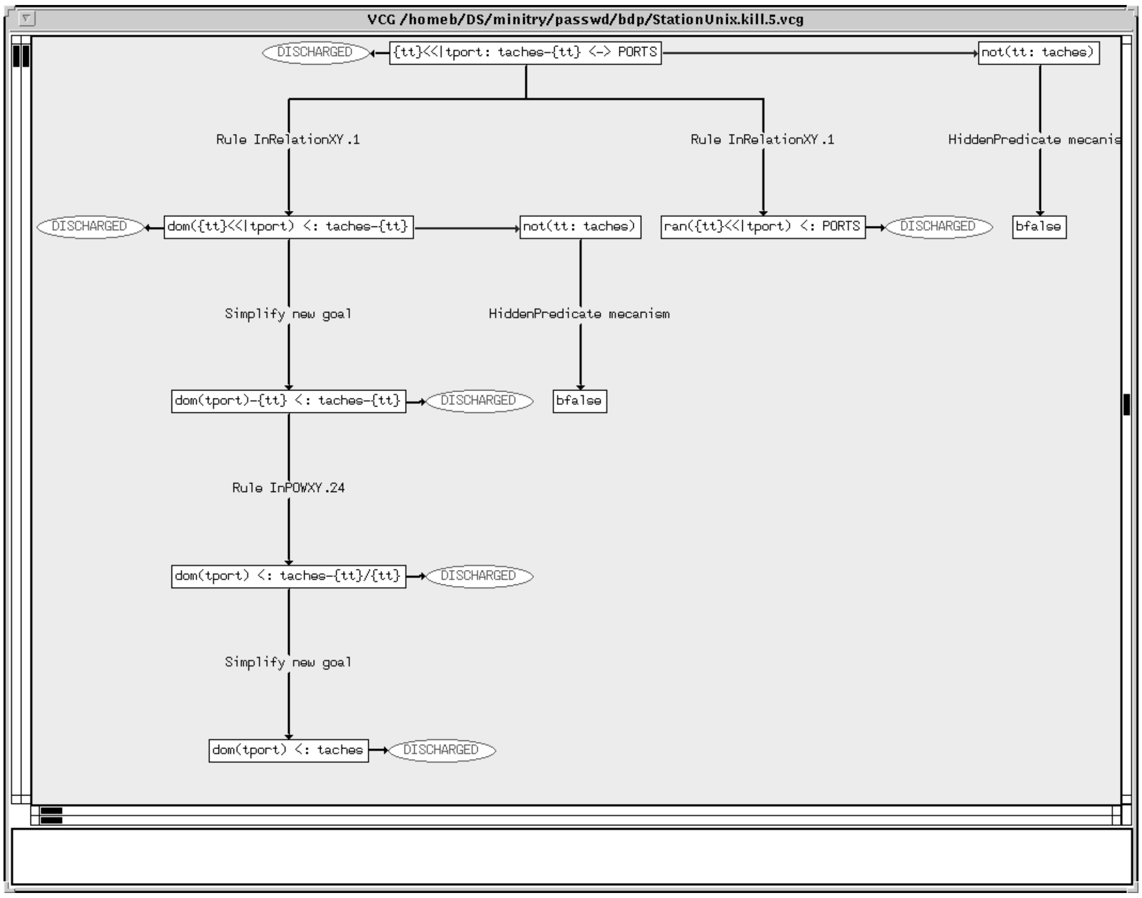
The first of the two sub-goals is discharged. We now go to the second one:

```
  By applying atomic rule InPOWLeavesXY.35,
       binhyp(ran(r): POW(b))
       =>
       ran(a<<|r): POW(b)
  the goal ran({tt}<<|tport) <: PORTS is discharged.

  End of trace

Current PO is kill.5
    Proved saved Proved
    Goal
        "‘Check ...4’" => {tt}<<|tport: taches-{tt} <-> PORTS
End
```

The demonstration is completed. This proof tracing enables us to understand the performed demonstrations but it is not a mathematical writing-up for the following reasons:

- Preliminary process on hypotheses is not traced. In our example, the $\mathsf{ran}(tport) \in \mathbb{P}(PORTS)$ hypothesis was used. This hypothesis does not come directly from the B component; it is a hypothesis derived by the preliminary process that was not traced.
- Useless attempt are present in the tracing.
- Tracing starts at the initial goal and splits it up in the application order of rules. On the contrary, a mathematical demonstration starts at the hypotheses and reaches the conclusion by successive demonstrations of intermediate results.

Tracing mode is nonetheless crucial to analyze a proof. It is also possible to display a proof tracing in a graph: after producing a tracing in a file (`File` mode of the `pr` previous command), use the *Show Proof Tree* option in the *Show/Print* menu in the Prover global situation window. The display created for our example is as follows (Caution: on some versions, the \ are not displayed):



6.7 Proof recipes

There is a number of methods adapted to proof situations that may facilitate success. These often consist in knowing how to conduct the proof towards Prover mechanisms to avoid doing the proof by yourself. You can always improve on such a set of methods. With some experience, the user will build up his/her own techniques. In this chapter, we will try and group the highest number of efficient recipes.

6.7.1 Situation commands

Selecting the 'right' command to progress in a proof is not given by systematic laws - otherwise these would have been included in the Automatic Prover strategies. We thus have to make with lists of commands to be considered according to the various possible situations. This is done in the following table. In the course of a proof, the user can try and identify the current situation in the left column then try the corresponding commands.

searching for an intuitive situation	`sh` (*SearchHypothesis*), `rp` (*showReducedPo*)
to try first	`pr` (*Prove*), `pp` (*PredicateProver*) specially for a set or functional goal
missing key hypothesis	`ah` (*AddHypothesis*), `eh` (*useEqualityinHypothesis*), new process of hypotheses in the Prover (refer section 6.7.2), applying rewriting rules.
the intuitive demonstration is done by cases	`dc` (*DoCases*)
what is to be done is expressed by a simple rule	`sr` (*SearchRule*), `vr` (*Validation of Rule*), `ar` (*ApplyRule*)
hypotheses should simplify each other	replay these hypotheses conjunction in the Prover (refer section 6.7.2)
existential goal of the form: $\exists x.P$	`se` (*SuggestforExists*)
contradictory hypotheses	`fh` (*FalseHypothesis*)
load a hypothesis in the Automatic Prover	`dd` (*Deduction*)
goal of the form: `not(p)`	`ct` (*Contradiction*)
the demonstration to be done look like an other one	`te` (*TryEverywhere*)
key hypothesis of the form $\forall x.(P(x) \Rightarrow Q(x))$	`ph` (*ParticularizeHypothesis*)
the $P \Rightarrow Q$ hypothesis is unexploited	`mh` (*ModusponensonHypothesis*)
before any risky command	`sw` (*SaveWithoutquestion*)

6.7.2 Replay hypotheses in the Prover

The *AddHypothesis* : `ah(h)` command is often used with the h parameter corresponding literally to an existing hypothesis. In this case, the result sought is indeed not to add the hypothesis but simply to unload it in the goal that becomes $h \Rightarrow B$, B, being the current goal. At the next call to the Automatic Prover, h will receive a privileged treatment from the local hypotheses. It is in fact a means to attract the Prover attention on this hypothesis.

This is specially useful when h is a derived hypothesis. Derived hypotheses are those produced after a normalization by forward rules, such as:

$$a = \mathsf{FALSE} \ \wedge \ a = \mathsf{bool}(P) \ \Rightarrow \ \neg(P)$$

This rule produces the $\neg P$ derived hypothesis if $a = \mathsf{FALSE}$ and$a = \mathsf{bool}(P)$ are hypotheses.

Hypotheses thus created do not benefit from the same treatment as the others - for efficiency reasons: loop risks.

It is then at times sufficient to replay a hypothesis in the Prover to succeed directly with a demonstration. The *AddHypothesis* command is optimized for such a case: `ah(h)` if h is already a hypothesis directly creates the $h \Rightarrow B$ goal without asking for a demonstration of h. Remember to replay hypotheses with the Prover in the following cases:

- when an important equality seems to be a derived hypothesis. Be specially careful with equalities created by the instantiation of $p \Rightarrow a = b$. You have to replay the new $a = b$ hypothesis in the Prover.

- when several predicates are simplified among themselves (for example: $\neg p \wedge \forall x.(x \in EE \wedge p \Rightarrow q)$) replays in the Prover the conjunction of these predicates. This strategy allows you to trigger the conjunctions / disjunctions simplification mechanism as it is started on each new goal.
- when a hypothesis whose normalization is not completed could lead to a successful demonstration (example : $\neg\neg P$ not simplified).

The Interactive Prover interface is optimized to enable a fast selection of an hypothesis to replay it in the Prover:

1. Place the cursor to the left of the hypothesis selected in the hypotheses window and drag it towards the bottom to darken an area beginning with this hypothesis;
2. Press the third button of the mouse; the selection is automatically framed on the selected hypothesis;
3. Point the *Add Hyp* button (on window top) and press *OK*: the desired `ah` command is automatically generated.

6.7.3 Instanciate $p \Rightarrow q$ if p is 'nearly' in hypothesis

If $p \Rightarrow q$ is in hypothesis and if a hypothesis is equivalent to p except for normalization, then the Prover would not have produced q because of the form difference on p. To do this, we advise the following method:

- add the p hypothesis as it appears in $p \Rightarrow q$.
- demonstrate this hypothesis. This proof should be performed easily, the Prover starting by bringing back p in its hypotheses form.
- the B initial goal becomes $p \Rightarrow B$; do `dd` to load p in hypothesis as it is.
- use `ph` to generate q

6.7.4 Think "ah" instead of forward rules

'Front' rules are specific rules enabling to generate new hypotheses from existing ones. For example, the $a \in \{b\} \Rightarrow a = b$ rule can generate the $xx = 3$ hypothesis from the $xx \in \{3\}$ hypothesis. In this Manual, we shall not study these rules that are more seldom used in interactive mode and easily create loops.

When a forward rule seems lacking - that is when a hypothesis derived from a simple rule would be desirable - it is often better to add this hypothesis with *AddHypothesis*. In most cases, the Prover should succeed in demonstrating this hypothesis that simply derives from current hypotheses: it is enough to load it with `dd` or `pr`. In the previous example, if by any chance the $xx = 3$ hypothesis was missing although $xx \in \{3\}$ was in hypothesis, we would only have to add $xx = 3$ with an *AddHypothesis* command.

The specific use of a rule is thus avoided - saving the time to search for it.

6.7.5 Normalization problems (brackets)

It often happens that the user wishes to make transformations requiring to go through forms that are contrary to the Prover normalizations. There could then be a kind of conflict between the user and the Automatic Prover - each one doing at the next step what the other has already done. For example, let us assume that the user wants to transform this hypothesis:

$$\mathsf{dom}(gg \mathbin{\lhd\!\!+} \{xx \mapsto aa\} \mathbin{\lhd\!\!+} \{xx \mapsto bb\}) = EE$$

We assume that the simplification of the two overloads on the same element was not done because the Prover did not group the right two terms. The user has found the following rule with a *SearchRule* command:

$$\{a \mapsto b\} \mathbin{\lhd\!\!+} \{a \mapsto c\} == \{a \mapsto c\} \quad \text{(SimplifyRelOveXY.13)}$$

But he/she cannot apply it as the implicit bracketing of the concerned hypothesis does not allow it. Instead of searching for another rule to modify the brackets, we recommend the following method:

- `ah(` $\mathsf{dom}(gg \mathbin{\lhd\!\!+} (\{xx \mapsto aa\} \mathbin{\lhd\!\!+} \{xx \mapsto bb\})) = EE$ `)`
- `pr`: immediate demonstration as the Automatic Prover rewrites the hypothesis under the actual form in hypothesis.
- `ar(SimplifyRelOveXY.13,Goal)`: note the application in 'Goal', the added hypothesis being still in the goal. This mode is restricted to rewriting rules.

Thus a transformation that requires avoiding Prover normalizations can be performed.

6.8 Traps to avoid

Unfortunately using the Interactive Prover has a few pitfalls we are going to describe.

6.8.1 Proofs by cases control

After each call to the Automatic Prover, you must check if the proof was not placed in a useless system of multiple cases. If this happens you might have to repeat the remainder of the demonstration in several proof paths. This is time-consuming.

To control the number of cases, it is more useful to use an admission rule (refer to section 6.6.2). When the Automatic Prover starts improper proofs by cases, use the `pr(Red)` command instead of `pr`, this mode dedicated to force 0 prevents proofs.

6.8.2 User rules numbers

Rules in a user rule file must not be deleted when there are useful rules after them in the same theory. These rules do change order sequence, inducing offsets when addressing them with *ApplyRule*.

For example, let us assume that a .pmm file contains three rules:

```
THEORY MyRules IS
  r1;
  r2;
  r3
END
```

If the user notices at the end of the component proof that `r1` is useless, he/she can be tempted to delete it. But in this case the commands that concern the third rule (`ar(MyRules.3,...)`) are not accepted (`MyRules.3 message : no such rule`) and those concerning the second rule no longer use the right rule. These commands can have been saved as proof obligation demonstrations and these are considered as proven. We only notice the problem if these proof obligations status is returned unproved and if we try to replay the proof.

It is for this reason always judicious to **Replace in an unproved status all components of a proof obligation and to replay all demonstrations**. To do this, use the `Unprove` option in the `Prove` menu in the main window then the `Prove (replay)` option in the same menu. This must be done when modification on the user rules become important and in any case at the end of a component proof. This precaution is useless only when no user rule has been used.

It is nonetheless desirable to be able to delete a useless rule to limit the user rules validation task. We advise you to replace the rule with a non-applicable one. For example:

```
THEORY MyRules IS
  empty_rule;            /* places 1 and 2 empty */
  empty_rule;
  (a - a/b) == (a mod b)  /* rule number 3 */
END
```

In the above example, the word `empty_rule` cannot coincide with a valid mathematical goal.

6.8.3 Changing forces during a proof

A user demonstration is usually performed in force 0 (refer section 6.1). It might nonetheless happen that during an interactive demonstration the user has the intuition that the PO difficulty is related to a specificity of one of the Prover forces - refer to section 6.6.4. You must then try that force with a simple call to the Automatic Prover - without loosing the interactive demonstration in case of a failure.

The `ff` command enables to be replaced in the same position with a different force. For example, if the present demonstration is:

```
Force(0) &
  pr &
    ar(OrderXY.63,Once) &
      pr &
        Next
```

At this step, the `ff(1)` command indicates we have to reload all the hypotheses in force 1 then replay all commands. The first call to the Automatic Prover has a very different result in force 1; this is why it is not likely that the exit goal will do to apply the OrderXY.63 rule. Thus the *ApplyRule* will be refused and the final status be:

```
Force(1) &
  pr &
    pr &
      Next
```

The *ApplyRule* command is lost! This is why you must save the command list before changing forces. The method is as follows:

- save demonstration in force 0 using `sw`
- return to the beginning of the demonstration with `re`
- move in higher force: `ff(x)` command
- prove in higher force: attempt a call to the Automatic Prover (`pr`), possibly reuse commands from previous demonstration(`st` command)
- if this fails, return to force 0 with `re`, `ff(0)` and repeat previous demonstration with `st(End)`

On the other hand, when a demonstration in force 1 is saved on a proof obligation, an interactive access to this obligation forces a change to force 1. You must then avoid to save in force 0 uselessly - this would slow down access without aim.

6.8.4 Loading problems

check loading messages:after loading the Automatic Prover and accessing the first unproved (*Next* command), it is possible to return to the beginning of the command window to check the messages of possible user file loading. These are the `Loading theory` messages. Possible errors are stated at this level.

Chapter 7

Useful indications for the proof

7.1 Carrying on taking care of the goal form

In most cases, the goal form is not sufficient to determine the type of demonstration to apply. These are often the local hypotheses that enable to determine it.

We can establish a classification of the goals and indicate commands to try for each of them:

- $\boxed{\exists x.P}$: in general, the command `se` (*SuggestForExist*) is compulsory, unless we have a goal of this form:

  $$\exists xx.(xx = 0)$$

  In the latter case, the commands `mp`/`pr`/`pp` succeed.

- $\boxed{P \Rightarrow Q}$: it is possible to use:
  - `dd` to load P directly in the hypotheses stack, without processing, (simplification, normalization) on them.
  - `dd(0)` to load P in the hypotheses stack, after normalization and application of force 0 simplifications.
  - `mp`, `pr` to simplify the goal and solve it,
  - `pp` to discharge it,
  - `pp(rp.0)` if Q can be demonstrated with only hypotheses P.

- $\boxed{\forall x.(P(x) \Rightarrow Q(x))}$: it is possible to use:
  - `mp`, `pr` to simplify the goal in $P(y) \Rightarrow Q(y)$ where y is a fresh variable, and solve it.
  - `pp` to discharge it,
  - `pp(rp.0)` if the goal can be demonstrated without additional hypotheses.

- $\boxed{A \vee B}$: it is possible to use:
  - `mp`, `pr` to simplify the goal and solve it.
  - `pp` to discharge it .
  - `ar(SplitOr.1, Once)` to carry on the proof with $not(B) \Rightarrow A$,
  - `ar(SplitOr.2, Once)` to carry on the proof with $not(A) \Rightarrow B$.
- $\boxed{A \wedge B}$: it is possible to use:
  - `mp`, `pr` to simplify the goal (decompose it) and solve it.
- $\boxed{a \in E}$: it is possible to use:
  - `ss` to simplify set predicates,
  - `mp`, `pr` to simplify the goal, and discharge it.
  - `pp` to discharge it,
  - `pp(rp.0)` if the goal can be demonstrated without additional hypotheses.
- $\boxed{a = b}$: it is possible to use:
  - `ap`, `pr`, `mp`, `pp` to discharge it
  - `eh`, if a and b appear in other equalities
- $\boxed{a < b,\ a \leq b,\ a > b,\ a \geq b}$: it is possible to use:
  - `ap`, `pr`, `mp`, `pp` to discharge it
- $\boxed{bfalse}$: it is possible to use:
  - `fh(H)` where H is a contradictory hypothesis
- $\boxed{P(x)}$ where $x \in E$ and E is an enumerated set or reduced interval

  - `dc(x, E)` to try to demonstrate $P(x)$ for all the possible values of x.

All these indications are quite general and don't take into account the characteristics of every imaginable proof project. Don't forget to check that the proof obligation you want to demonstrate is true before starting on the interactive proof.

7.2 When and how to use the predicate prover

7.2.1 Reducing the number of hypotheses

The predicate prover has to be used when `mp` and/or `pr` failed. These three provers have partially common domains. On the opposite of `mp` or `pr`, `pp` only works with a reduced set of hypotheses.
The running time off pp is exponantional with the number of hypotheses. So, it is not

reasonable to use `pp` to demonstrate a proof obligation, especially if the component sees and/or imports other components.

The number of hypotheses has then to be reduced. Three methods are available:

- selection of the hypotheses having a common identifier with the goal.
  The main limitation is that we can't filter precisely the selection and if one of the goal identifier appears in many hypotheses, `pp` will turn out inefficient.
  Example: `pp(rp.1)` to select the hypotheses which have a common identifier with the goal.
- selection of hypotheses according to their origin.
  The hypotheses are selected by group without any possibility to be removed.
  Example: `pp(rp(loc+inv))` to select local hypotheses and the component invariant.
- manual selection of hypotheses.
  When the hypotheses are selected by the command `ah(H)`. The hypotheses are already present in the hypotheses stack. The goal B becomes

  $$H \Rightarrow B$$

  When all the hypotheses are included in the goal in this manner (the goal is as follows

  $$H_n \Rightarrow (H_{n-1} \Rightarrow (... \Rightarrow B)...))$$

  use `pp(rp.0)`.

It is important not to forget to include the well definition hypotheses of the lemma to demonstrate. These hypotheses are often necessary to demonstrate the lemma.

For example, if the goal contains the expression

$$f(x)$$

the following 2 hypotheses should be added

$$x \in dom(f)$$
$$f \in A \nrightarrow B$$

7.2.2 Computation time limit

It is possible to customize a time out for `pp` (maximum time dedicated to a `pp` run). A good time out is 10 s. This value enables, through the command `te` (*TryEveryWhere*), to quickly test `pp` on several proof obligation. Using it in a systematic way with a higher value (for example 300 s) is still possible but the gain in proof rate will probably remain modest. This time out is not applied in automatic proof when using the User_Pass.

7.2.3 Using pp wisely

Should you use pp or add a rule?
There is no absolute answer to this question. It will be up to the user to decide at last which one of the two options to choose, according to his experience and his preferences. However, here are some hints for each approach:

- a pure interactive demonstration (with no user rules added) does not require additional checking, but can induce the generation of numerous proof steps, leading to an important proof lentgh. Remind that an interactive demonstration may be lost when some changes of the B model induce a modification of the proof obligation.
- adding a rule often saves time in interactive proof, especially if the rules base doesn't contain the rules necessary to prove the goal. This added rule should be carefully demonstrated (and should be verified by another person), in order to avoid demonstrating false proof obligations and thus, invalidate the current development.

7.2.4 Using proof strategies

`pp(rp.0)` can be used in the `User_Pass`. For that, edit the `PatchProver` file (in the pdb project directory) or the `Pmm` file of your component, add a `User_Pass` theory containing one proof strategy per rule.

Given, for example, the component *tests*, which associated `Pmm` file contains the following theory:

```
THEORY User_Pass IS
    mp;
    pp(rp.0) ;
    dd(0) & ap
END
```

This theory defines the interactive demonstrations the user wants to be tried. We can see that `pp(rp.0)` will be tried after `mp`.

Each line, from the top to the bottom of the theory, will then be applied as long as non-demonstrated proof obligation are left.

The user pass is launched, by selecting the button *Prove - User_Pass.* The following messages are displayed:

```
Loading theory User_Pass
Proving tests
  Proof pass User_Pass.1, still 8 unproved PO
    clause Initialisation
    ---
    clause AssertionLemmas
    -++-
    clause op
    -

  Proof pass User_Pass.2, still 6 unproved PO
    clause Initialisation
    ---
    clause AssertionLemmas
    +-
    clause op
    -

  Proof pass User_Pass.3, still 5 unproved PO
    clause Initialisation
    ++-
    clause AssertionLemmas
    +
    clause op
    -
```

We can see that:

- `mp` (`User_Pass.1`) has demonstrated 3 proof obligations,
- `pp(rp.0)` (`User_Pass.2`) has demonstrated 1 proof obligagation,
- `dd(0) & ap` (`User_Pass.3`) has demonstrated 2 proof obligation,

On the other hand, the predicate prover time out cannot be used anymore in these conditions.

However, it is possible to determine the computing time of the predicated prover and to prevent it from consuming too much time on some proof obligation. For that, after launching the automatic proof *Prove - User Pass*, the user must select the button *Interrupt - Next PO* each time he thinks the proof time takes too long.

7.3 User rules application

Here are some hints for the application of a user rule:

- $\boxed{\text{do not forget to normalize the rules:}}$
  - The rules of Pmm files are normalized by the prover when they are loaded,
  - the rules of PatchProver files have to be normalized by the user.
  - $a < b$ is normalized by the prover in $a + 1 \leq b$. Thus the guard $\mathsf{btest}(a < b)$ will be rewrited $\mathsf{btest}(a + 1 \leq b)$ and will never succeed.
- in the case of complex expressions, $\boxed{\text{do not hesitate to overparenthesise}}$ your rule terms.
- check that you $\boxed{\text{use only jokers}}$ (one letter identifiers).
  The rule:

  $\mathsf{binhyp}(xx = 0)$
  $\Rightarrow$
  $xx \ \mathsf{mod}\ 2 = 0$

  will be applied only if the goal is strictly $xx \ \mathsf{mod}\ 2 = 0$,but not if it is $yy \ \mathsf{mod}\ 2 = 0$.
- check that you $\boxed{\text{do not introduce never instanciated jokers}}$ in your rule. In practice, all the jokers appearing in a Backward rule consequent are instanciated. Make sure that, in the case of a goal replacement by an equivalent one, the generated goal is fully instanciated.

  For example, the rule:

  $\mathsf{binhyp}(H)\ \wedge$
  $(h\ \Rightarrow\ B)$
  $\Rightarrow$
  B

  will produce, for the goal

  $xx = 0$

  the derived incoherent goal

  $h\ \Rightarrow\ xx = 0$

  Other example, the rule:

  $\mathsf{binhyp}(b)$
  $\Rightarrow$
  B

  is false because the mathematical associated lemma

$$(b \Rightarrow B)$$

is false.

- if the goal form is as follows

  $$H \Rightarrow Q$$

  and you want to use the hypotheses H to solve Q, first $\boxed{\text{load them in the hypotheses stack}}$, using the command `dd` or `dd(0)`.

- a $\boxed{\text{backward rule}}$ has the following form:

  $$\begin{array}{l} guard(s) \ \wedge \\ subgoal(s) \\ \Rightarrow \\ goal \end{array}$$

  which means that *goal* is changed in *subgoal* if the *guard* is true.
  guard(*s*) and *subgoal*(*s*) are optional. For example:

  $$a - a = 0.$$

- a $\boxed{\text{rewritting rule}}$ has the following form:

  $$\begin{array}{l} guard(s) \ \wedge \\ subgoal(s) \\ \Rightarrow \\ formula1 == formula2 \end{array}$$

  which means that $formula1$ is replaced by $formula2$ if the *guard* is true and if the *subgoal* is demonstrated.
  guard(*s*) and *subgoal*(*s*) are optional. For example:

  $$a + 1 - 1 == a.$$

The guards are used to reduce the rule application to the only cases where the guards hold.

The most useful guards are:

- binhyp(H): true if H is in the hypothesis stack,

- btest(a *op* b): true if a *op* b is true (*op* is one of the operators $=$,$<$,$>$,$\leq$,$\geq$ and a, b are indentifiers or literal integers)

- bnot(G): true if G is false.
  Warning! for $bnot(btest(a = b))$, the test is only performed on the names of a and b. This guard will therefore succeed if a and b are variables with different names, despite the fact that those variables have the same value.
  We could have for example:

$$\mathsf{binhyp}(a = 1) \ \wedge \ \mathsf{binhyp}(b = 1)$$

considered as true. So be careful.

- $\mathsf{bnum}(a)$: true if a is a non negative integer smaller than $MAXINT$.
- $\mathsf{bgoal}(G)$: true if the current goal has the form G.
- $x\backslash P$: true if x does not have a free occurrence in P.
  $x\backslash(x + 3)$ is false
  $x\backslash(\forall x.(x \in E \ \Rightarrow \ P(x)))$ is true
- $\mathsf{blvar}(Q)$: Q is instanciated with the list of currently quantified variables.

These last two guards are required for the redaction of non-atomic rewriting rules. Indeed, be careful not to capture variable.
For example, if we use a binhyp, we must check that the instanciated variables are non-free in the list of quantified variables by using the guards $\mathsf{blvar}(Q)$ and $Q\backslash x$.
For example, given the false proof obligation:

$$\begin{array}{l} xx \in NAT \ \wedge \\ xx = 0 \ \wedge \\ xx + 1 = 1 \ \wedge \\ xx + 2 = 2 \\ \Rightarrow \\ \forall xx.(xx + 1 \in \mathbb{N} \ \Rightarrow \ xx + 2 \in \mathbb{N}) \end{array}$$

and the rule

$$\begin{array}{l} \mathsf{binhyp}(a + b = c) \\ \Rightarrow \\ (a + b == c) \end{array}$$

which replaces $a + b$ with c.
The application of this rule on the goal will change it in

$$\forall xx.(1 \in \mathbb{N} \ \Rightarrow \ 2 \in \mathbb{N})$$

which is true. The error comes from the confusion between the variable xx in the hypotheses stack and the dummy variable xx.
The rule should be guarded as follows:

$$\begin{array}{l} \mathsf{binhyp}(a + b = c) \ \wedge \\ \mathsf{blvar}(Q) \ \wedge \\ Q\backslash(a, b, c) \\ \Rightarrow \\ (a + b == c) \end{array}$$

Regarding this type of rule, we must be careful.
For exampe, the rule

$\mathsf{binhyp}(a = b)\ \land$
$\mathsf{blvar}(Q)$
$\Rightarrow$
$a == b$

is false. Indeed, symbol \$ is considered as an operator. So aa\$0, which is a perfect acceptable indentifier, can be decomposed in aa, \$ and 0.
Let us assume that the following hypotheses are in the hypotheses stack:

$aa = 1$
$0 = xx$

By applying the above rule twice, we may change the identifier aa\$0 in 1\$$xx$!

In this case, it is required to handle the full predicate. For example, the rule

$\mathsf{binhyp}(a = b)\ \land$
$\mathsf{blvar}(Q)$
$\Rightarrow$
$(0 \leq \mathrm{a} == 0 \leq \mathrm{b})$

is perfectly valid.

7.4 Adding user's rules

It is possible that a rule cannot be applied when using the `ar` command. The search of rules that can be applied on the goal (`sr` command) returns the rules that can POSSIBLY be applied. Then you should check that these rules guards are true.

A rule cannot be applied because:

- it is badly normalized (user's rule). Then it has to be corrected.
- the guards are not true
- a hypothesis does not exist.
- the goal does not have exactly the rule consequent form:
  - check if another rule maybe applied,
  - the goal can be rewrited with another acceptable form, that will enables the rule application.
    Let us imagine that the goal contains the expression E whereas the rule expects the expression E'. It is possible to add the hypothesis $E = E'$ (`ah(E=E')`).
    This equality can be demonstrated by `pp(rp(0))`, `ss`, `mp` or `pr`. Once this equality is demonstrated, it is loaded in the hypotheses stack (`dd` command).
    This equality is then applied on goal (`eh(E)`). The rule can then be applied.
- check that you didn't made an error when adding a hypothesis (misspelling). In that case, go back to this hypothesis addition, that must be corrected, then return to where you stopped in the interactive demonstration.

Systematically check the good writting of the rules you add. For that, use the command `sr(usertheory,a)`. All the user's rules present in the *usertheory* theory will be then displayed.

For example, command `sr(tt,a)` displays all the rules of the *tt* theory rules:

```
PRI > sr(tt,a)
Searching in tt rules with filter
    consequent should contain a
Starting search...
Rule list is
    tt.1
        binhyp(s: seq(T))
        =>
        (size(s) = 0 == s = {})
End of rule list
```

Do not forget that the added rules in a `pmm` during interactive proof are taken into account only after the `pc` command had been performed.
In the same way, do not forget that if you delete a rule in the middle of a theory, the rules order will be modified and then, command `ar(regle.n, Once)` may be not applied due to the rules shift, the rule *rule.n* does no longer refer to the right rule. Furthermore, if you add/remove/change one or several rules of your theory, the `ar(user_theory)` command cannot be applied any longer.

In general, when modifying your user's theories, it is advised to unprove the considered component, then to perform a *Prove - Replay*, to check that there is no proof regression.

For the `PatchProver` file, do not forget that if you modify it during a proof phase, it won't be loaded even if you quit the interactive prover. For that, quit the project then reopen it, and restart the interactive prover.

A rule should be added when provers and solvers don't succeed in solving or simplifying the current goal. It may be a simple logic proposition containing complex expressions that will disturb `pp`.
In that case, we add a rule corresponding to the goal, where complex expressions are replaced with jokers. The rule must always be simple as it will then have to be demonstrated.

When adding rules, ask yourself some questions regarding the rule writting:

- Is my rule too specific?
- Should I write a complex rule or several easier ones ?
- Is there a rule of the basis rule that can be used in a different form?

7.5 Ease the proof by adding information in the B model

It is possible to add information in the B model that can ease the proof work. It concerns the ASSERTIONS, PROPERTIES and ASSERT clauses.

These three clauses have various scopes:

- substitution, for ASSERT,
- component for ASSERTIONS and PROPERTIES,

PROPERTIES characterizes the constants used (typing + properties expressions).
ASSERTIONS characterizes the component's variables (an assertion has to be demonstrated under the hypothesis that the invariant and the previous assertions are true).

There are various manners to express these properties. Some of them are easily demonstrable, whereas others are less easy to demonstrate.
In fact, adding an assertion (A) corresponds to systematically add the A hypothesis to all the component proof obligations.

Adding such properties is a consequence of the proof work and requires a good experience of the prover and its functioning, to avoid adding assertions that would be useless as they would have been badly expressed or would not change the proof path.

You must therefore be careful when modifying these assertions as it is possible to induce proof regressions.

Let's assume that the goal P of a proof obligation is being demonstrated interactively under H hypotheses and A assertions. what's happening if a B assertion is added?

Logically, if

$$H \ \wedge \ A \ \Rightarrow \ P$$

is true, we are bound to have

$$H \ \wedge \ A \ \wedge \ B \ \Rightarrow \ P$$

The proof obligation is then always demonstrated and its interactive demonstration is saved.

Now, if an assertion is considered as useless and then deleted, given the B assertion, and if we had

$$H \ \wedge \ A \ \wedge \ B \ \Rightarrow \ P$$

we cannot conclude anymore on

$$H \wedge A \Rightarrow P$$

The proof obligation generator considers that this proof obligation is not proved anymore but does not delete the saved interactive demonstration. In that case, the automatic prover starting up in *Prove - Replay* mode reproves this proof obligation.

Let's assume that this assertion deleting is coupled with other component modifications inducing a modification of the proof obligation number for the considered component. In such a case, the previous interactive demonstration can be considered as lost as it is from now, associated to another proof obligation (due to the proof obligation's shifting) which has few chances to be demonstrated by this interactive demonstration.

7.6 Using the Do Cases command

The `dc` command is required when:

- $P(x)$ must be demonstrated with $x \in E$ (E is an intervall) (command `dc(x,E)`)
- $P(x)$ must be demonstrated under the hypotheses $(A \Rightarrow Q(x)) \wedge (not(A) \Rightarrow R(x))$, and Q and R enable to discharge $P(x)$. `dc(A)`

This command must be applied if `mp`/`pr`/`pp` didn't or wouldn't succeed in discharging. Warning ! The `pr` command can induce proof per case, according to the goal and to some hypotheses. These proof per case may not be relevant and require to prove sereral times the same goal (due the fact that the pr heureustics are general and can be sub-optimal in some cases).

Example: given the proof obligation

```
        STATES = {E0, E1, E2} &
        xx: STATES &
        "‘Local hypotheses’" &
        xx = E0 => xx$1: {E0,E1} &
        xx = E1 => xx$1: {E0,E1,E2} &
        xx = E2 => xx$1: {E1,E2} &
        "‘Check that the invariant (xx: STATES) is preserved by
        the operation - ref 3.4’" &
        =>
        xx$1: STATES
```

We first perform `mp` in order to load the hypotheses in the stack hypotheses. The goal becomes:

```
PRI > mp
Starting Prover Call
        xx$1: STATES
```

When examining the hypotheses, we see that the domain of $xx\$1$ depends on the xx value. Then a proof per case for xx describing all the values of the enumerated set $STATES$ has to be performed.

```
PRI > dc(xx,STATES)
Do Cases on Enumerated {E2,E1,E0}
        xx = E2 => xx$1: STATES
```

The first goal $xx \in STATES$ has obviously been demonstrated by the prover. These three cases are now going to be generated. The first of them has to be discharged:

$$xx = E2 \;\Rightarrow\; xx\$1 \in STATES$$

We can perform an automatic prover call (`mp` or `pr`). We can also perform a predicates prover call (`pp(rp.0)`), after adding a local hypothesis

$$xx = E2 \ \Rightarrow \ xx\$1 \in \{E1, E2\}$$

with the command `ah`:

```
PRI > ah(xx$1: {E1,E2})
Starting Add Hypothesis
        xx$1: {E1,E2} => xx$1: STATES
```

The predicates prover call demonstrates this first goal and then the prover continues on the second case.

```
PRI > pp(rp.0)
Starting Predicate Prover Call
Proved by the Predicate Prover
        xx = E1 => xx$1: STATES
```

We add the following hypothesis

$$xx = E1 \ \Rightarrow \ xx\$1 \in \{E0, E1, E2\}$$

then, the predicates prover is called.

```
PRI > st
Next step ah(xx$1: {E0,E1,E2})
Starting Add Hypothesis
        xx$1: {E0,E1,E2} => xx$1: STATES

PRI > pp(rp.0)
Starting Predicate Prover Call
Proved by the Predicate Prover
        xx = E0 => xx$1: STATES
```

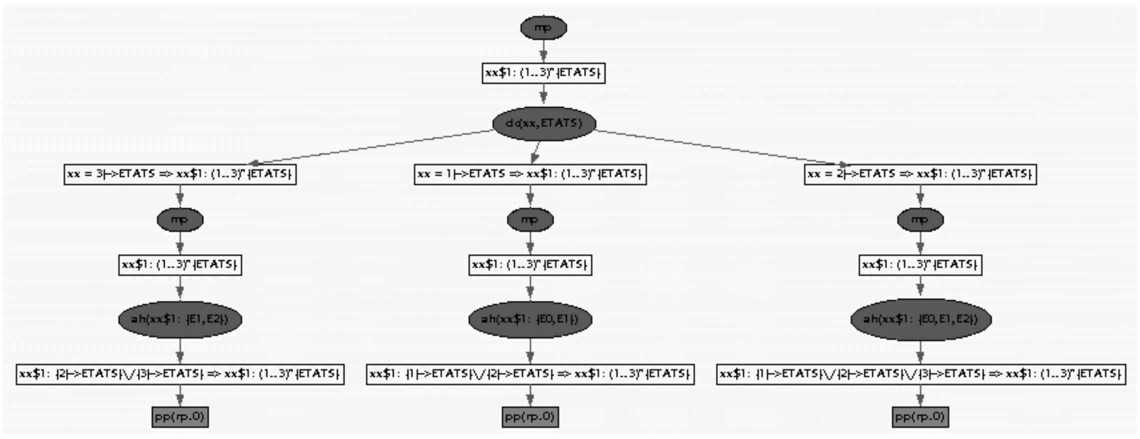
The second subgoal is proved. The third one has therefore to be demonstrated. Just add the hypothesis

$$xx = E0 \ \Rightarrow \ xx\$1 \in \{E0, E1\}$$

then call again the predicates prover in order to demonstrate the last goal.

```
PRI > ah(xx$1: {E0,E1})
Starting Add Hypothesis
        xx$1: {E0,E1} => xx$1: STATES

PRI > pp(rp.0)
Starting Predicate Prover Call
Proved by the Predicate Prover
```

The proof tree of the demonstration is as follows:


mp
xx$1: (1..3)*{ETATS}
dd(xx,ETATS)
xx = 3|->ETATS => xx$1: (1..3)*{ETATS}
xx = 1|->ETATS => xx$1: (1..3)*{ETATS}
xx = 2|->ETATS => xx$1: (1..3)*{ETATS}
mp
mp
mp
xx$1: (1..3)*{ETATS}
xx$1: (1..3)*{ETATS}
xx$1: (1..3)*{ETATS}
ah(xx$1: {E1,E2})
ah(xx$1: {E0,E1})
ah(xx$1: {E0,E1,E2})
xx$1: {2|->ETATS}\/{3|->ETATS} => xx$1: (1..3)*{ETATS}
xx$1: {1|->ETATS}\/{2|->ETATS} => xx$1: (1..3)*{ETATS}
xx$1: {1|->ETATS}\/{2|->ETATS}\/{3|->ETATS} => xx$1: (1..3)*{ETATS}
pp(rp.0)
pp(rp.0)
pp(rp.0)

7.7 Application : first example

Given the components

```
MACHINE
    M1
VISIBLE_VARIABLES
    tab
INVARIANT
    tab: 0..7 --> 0..1
INITIALISATION
    tab := (0..7) * {0}
OPERATIONS
    op =
        BEGIN
        tab: (
            tab: 0..7 --> 0..1 &
            tab(0) = 0 &
            tab(1) = 1 &
            tab(2) = 1 &
            tab(3) = 0  &
            tab(4) = 1 &
            tab(5) = 0 &
            tab(6) = 1 &
            tab(7) = 0
            )
        END
END
```

and

```
IMPLEMENTATION
    M1_i
REFINES
    M1
INITIALISATION
    tab := (0..7) * {0}
OPERATIONS
    op =
        BEGIN
            tab(0) := 0 ;
            tab(1) := 1 ;
            tab(2) := 1 ;
            tab(3) := 0 ;
            tab(4) := 1 ;
            tab(5) := 0 ;
            tab(6) := 1 ;
            tab(7) := 0
        END
END
```

After proving the two components in force 0, the project status is as follows one:

```
Project status
+-----------+----+-----+-----+-----+-----+-----+-----+-----+----+-----+
| COMPONENT | TC | POG | Obv | nPO | nUn | %Pr | BOC |  C  | Ada | C++ |
+-----------+----+-----+-----+-----+-----+-----+-----+-----+----+-----+
| M1        | OK | OK  |   3 |   2 |   0 | 100 |  -  |     |     |     |
| M1_i      | OK | OK  |   7 |  18 |   8 |  55 |  -  |  -  |  -  |  -  |
+-----------+----+-----+-----+-----+-----+-----+-----+-----+----+-----+
| TOTAL     | OK | OK  |  10 |  20 |   8 |  60 |  -  |  -  |  -  |  -  |
+-----------+----+-----+-----+-----+-----+-----+-----+-----+----+-----+
```

After starting up the interactive prover, the **gs** command displays the form of the proof obligations to be demonstrated:

```
PO9 Unproved    tab$1<+{0|->0}<+{1|->1}<+{2|->1}<+{3|->0}<+{4|->1}
                <+{5|->0}<+{6|->1}<+{7|->0}: 0..7 +-> 0..1
PO11 Unproved   (tab$1<+{0|->0}<+{1|->1}<+{2|->1}<+{3|->0}<+{4|->1}
                <+{5|->0}<+{6|->1}<+{7|->0})(0) = 0
PO12 Unproved       (tab$1<+{0|->0}<+{1|->1}<+{2|->1}<+{3|->0}<+{4|->1}
                <+{5|->0}<+{6|->1}<+{7|->0})(1) = 1
PO13 Unproved       (tab$1<+{0|->0}<+{1|->1}<+{2|->1}<+{3|->0}<+{4|->1}
                <+{5|->0}<+{6|->1}<+{7|->0})(2) = 1
PO14 Unproved       (tab$1<+{0|->0}<+{1|->1}<+{2|->1}<+{3|->0}<+{4|->1}
                <+{5|->0}<+{6|->1}<+{7|->0})(3) = 0
PO15 Unproved       (tab$1<+{0|->0}<+{1|->1}<+{2|->1}<+{3|->0}<+{4|->1}
                <+{5|->0}<+{6|->1}<+{7|->0})(4) = 1
PO16 Unproved       (tab$1<+{0|->0}<+{1|->1}<+{2|->1}<+{3|->0}<+{4|->1}
                <+{5|->0}<+{6|->1}<+{7|->0})(5) = 0
PO17 Unproved       (tab$1<+{0|->0}<+{1|->1}<+{2|->1}<+{3|->0}<+{4|->1}
                <+{5|->0}<+{6|->1}<+{7|->0})(6) = 1
```

Seven proof obligations have the same form. The rules search matching with

$$(f < +g)(x) = c$$

the command `sr(All, ((f<+g)(x)=c))` doesn't not find applicable rules. Rather than performing seven successive interactive proofs, the addition of a simplification rule may be considered. The M1_i. pmm file now contains:

```
THEORY func IS
    bcall1(BackwardRule(func.1)) &
    bnum(a) &
    bnum(c) &
    bnot(btest(a=c)) &
    (f(c) = d)
    =>
    ((f<+{a|->b})(c) = d)
END
```

This rule should be compiled and loaded in memory using the `pc` command. We then check that this rule has successfully been loaded in memory using the SearchRule command:

```
PRI > pc
Loading theory func
PRI > sr(func,a)
Searching in func rules with filter
    consequent should contain a
Starting search...
Rule list is
    func.1      (Backward)
        bnum(a) &
        bnum(c) &
        bnot(btest(a = c)) &
        f(c) = d
        =>
        (f<+{a|->b})(c) = d
End of rule list
```

Now it can be used, especially in an extended automatic prover call. In this special case, the $func$ theory is used in addition of rules and mechanisms embedded in the prover. The trace mode of the prover is used in order to show the action of the $func.1$ rule.

Note that if the rule wasn't equiped with the trace system (term $bcall1(BackwardRule(func.1))$, its starting up would not be displayed in this mode.

```
PRI > pr(Tac(func), Ru.Goal.None)
  Starting Trace in mode Ru.Goal.None , NoFile , NoSimpl
Starting Prover Call
```

```
  After deduction, goal is now
        (tab$1<+{0|->0}<+{1|->1}<+{2|->1}<+{3|->0}<+{4|->1}
        <+{5|->0}<+{6|->1}<+{7|->0})(0) = 0
  By applying atomic rule func.1,
       bnum(a) &
       bnum(c) &
       bnot(btest(a = c)) &
       f(c) = d
       =>
       (f<+{a|->b})(c) = d
  the goal (tab$1<+{0|->0}<+{1|->1}<+{2|->1}<+{3|->0}<+{4|->1}
        <+{5|->0}<+{6|->1}<+{7|->0})(0) = 0  is now
       (tab$1<+{0|->0}<+{1|->1}<+{2|->1}<+{3|->0}<+{4|->1}
        <+{5|->0}<+{6|->1})(0) = 0
```

The rule has indeed been applied. The goal is simplified. The proof continues with the successive applications of this rule:

```
the goal (tab$1<+{0|->0}<+{1|->1}<+{2|->1}<+{3|->0}<+{4|->1}
         <+{5|->0}<+{6|->1})(0) = 0  is now
         (tab$1<+{0|->0}<+{1|->1}<+{2|->1}<+{3|->0}<+{4|->1}
         <+{5|->0})(0) = 0
the goal (tab$1<+{0|->0}<+{1|->1}<+{2|->1}<+{3|->0}<+{4|->1}
         <+{5|->0})(0) = 0  is now
         (tab$1<+{0|->0}<+{1|->1}<+{2|->1}<+{3|->0}<+{4|->1})(0) = 0
the goal (tab$1<+{0|->0}<+{1|->1}<+{2|->1}<+{3|->0}<+{4|->1})(0) = 0  is now
         (tab$1<+{0|->0}<+{1|->1}<+{2|->1}<+{3|->0})(0) = 0
the goal (tab$1<+{0|->0}<+{1|->1}<+{2|->1}<+{3|->0})(0) = 0  is now
         (tab$1<+{0|->0}<+{1|->1}<+{2|->1})(0) = 0
the goal (tab$1<+{0|->0}<+{1|->1}<+{2|->1})(0) = 0  is now
         (tab$1<+{0|->0}<+{1|->1})(0) = 0
the goal (tab$1<+{0|->0}<+{1|->1})(0) = 0  is now
         (tab$1<+{0|->0})(0) = 0
```

then the last goal is demonstrated by the prover:

```
Goal (tab$1<+{0|->0})(0) = 0  is discharged.
```

The demonstration is saved:

```
PRI > sw
PO op.11 saved
```

At last, as seven of the eight left proof obligations look the same, this demonstration is tried out (TryEverywhere command) on all the unproved proof obligations:

```
te(op.11)
Begin TryEveryWhere
-++++++
Summary
op.17 transformed   Unproved --> Proved,   pr(Tac(func))
op.16 transformed   Unproved --> Proved,   pr(Tac(func))
op.15 transformed   Unproved --> Proved,   pr(Tac(func))
op.14 transformed   Unproved --> Proved,   pr(Tac(func))
op.13 transformed   Unproved --> Proved,   pr(Tac(func))
op.12 transformed   Unproved --> Proved,   pr(Tac(func))
End TryEveryWhere
```

Let us consider the last unproved proof obligation:

```
tab$1<+{0|->0}<+{1|->1}<+{2|->1}<+{3|->0}<+{4|->1}
    <+{5|->0}<+{6|->1}<+{7|->0}: 0..7 +-> 0..1
```

Once again, no prover rule can efficiently simplify the goal. A new rule is added, after checking its validity using the proof rules tools:

```
PRI > vr(Back, (b: A+->B & f: A+->B => (f<+b): A+->B))
The rule was proved
```

This rule is then added to the M1_i.pmm file, in the simple theory this time:

```
THEORY simpl IS
    bcall1(BackwardRule(func.1)) &
    f: A +-> B & b: A +-> B
    =>
    f<+b : A +->B
END
```

This rule is loaded in memory:

```
PRI > pc
Loading theory func
Loading theory simpl
```

Check that this rule is successfully loaded in memory:

```
PRI > sr(simpl, a)
Searching in simple rules with filter
    consequent should contain a
Starting search...
Rule list is
    simpl.1      (Backward)
        f: A +-> B &
        b: A +-> B
        =>
        f<+b: A +-> B
End of rule list
```

The *simpl*.1 rule can now be used, especially in an extended automatic prover call. In this special case, the *func* theory is used in addition of rules and mechanisms embedded in the prover. The trace mode of the prover is used in order to show the action of the *func*.1 rule.

```
PRI > pr(Tac(simpl), Ru.Goal.None)

  Starting Trace in mode Ru.Goal.None , NoFile , NoSimpl
Starting Prover Call
  After deduction, goal is now
        tab$1<+{0|->0}<+{1|->1}<+{2|->1}<+{3|->0}<+{4|->1}
        <+{5|->0}<+{6|->1}<+{7|->0}: 0..7 +-> 0..1
  By applying atomic rule simpl.1,
       f: A +-> B &
       b: A +-> B
       =>
       f<+b: A +-> B
  the goal tab$1<+{0|->0}<+{1|->1}<+{2|->1}<+{3|->0}<+{4|->1}
       <+{5|->0}<+{6|->1}<+{7|->0}: 0..7 +-> 0..1  is now
       tab$1<+{0|->0}<+{1|->1}<+{2|->1}<+{3|->0}<+{4|->1}
       <+{5|->0}<+{6|->1}: 0..7 +-> 0..1
   and {7|->0}: 0..7 +-> 0..1
```

The goal will then be successively decomposed by the action of the *simpl*.1 rule and by the prover embedded rules..

7.8 Application : second example

Given the new M1_i component, replacing the previous one:

```
IMPLEMENTATION
    M1_i
REFINES
    M1
INITIALISATION
    tab := (0..7) * {0}
OPERATIONS
    op =
        BEGIN
            tab(0) := 0 mod 2 ;
            tab(1) := 1 mod 2;
            tab(2) := 1 mod 2;
            tab(3) := 0 mod 2;
            tab(4) := 1 mod 2;
            tab(5) := 0 mod 2;
            tab(6) := 1 mod 2;
            tab(7) := 0 mod 2
        END
END
```

The only difference according to the previous example is the appearance, only demonstrative here, of the modulo that prevents the previous demonstrations from succeeding. To get back to the previous example, we just need to add in hypotheses of all those proof obligations, the following predicates.

$$0 \text{ mod } 2 = 0$$
$$1 \text{ mod } 2 = 1$$

then to use these equalities to replace the left term with the right term in each proof obligations.

A solution consists in adding predicates in assertions:

```
ASSERTIONS
    0 mod 2 = 0;
    1 mod 2 = 1
```

These two assertions are easily demonstrable in interactive proof (`pp(rp.0)`).

The unproved proof obligation demonstration is performed by using these two equalities then by performing the interactive demonstration of the previous example.

The demonstration of the seven proof obligations having the same form is performed with only one TryEveryWhere command:

```
te((eh(0 mod 2) & eh(1 mod 2) & pr(Tac(func))), Replace.Loc.Unproved)
```

Then we check that these proof obligations are demonstrated:

```
Begin TryEveryWhere
-+++++++
Summary
op.25 transformed   Unproved --> Proved,   eh(0 mod 2) & eh(1 mod 2) & pr(Tac(func))
op.24 transformed   Unproved --> Proved,   eh(0 mod 2) & eh(1 mod 2) & pr(Tac(func))
op.23 transformed   Unproved --> Proved,   eh(0 mod 2) & eh(1 mod 2) & pr(Tac(func))
op.22 transformed   Unproved --> Proved,   eh(0 mod 2) & eh(1 mod 2) & pr(Tac(func))
op.21 transformed   Unproved --> Proved,   eh(0 mod 2) & eh(1 mod 2) & pr(Tac(func))
op.20 transformed   Unproved --> Proved,   eh(0 mod 2) & eh(1 mod 2) & pr(Tac(func))
op.19 transformed   Unproved --> Proved,   eh(0 mod 2) & eh(1 mod 2) & pr(Tac(func))
End TryEveryWhere
```

Chapter 8

Case studies

In this chapter we present a few interactive proofs. After each of these proofs, we will sum up the essence of the method used.

8.1 Simple proof by contradiction

We give here an example of an interactive proof that includes no user rule. The initial goal is **bfalse** which means that the proof can be true only by contradictory hypotheses. The Prover fails because he does not know how to set the contradictory hypothesis aside. We are to see how to indicate this to the Prover.

```
"‘Previous components invariants’" ∧
ens < ∈ NAT ∧
card(ens) ≤ 3 ∧
"‘Component invariant’" ∧
v1$1 ∈ ℕ ∧
v1$1 ≤ 2147483647 ∧
v2$1 ∈ ℕ ∧
v2$1 ≤ 2147483647 ∧
v3$1 ∈ ℕ ∧
v3$1 ≤ 2147483647 ∧
taille$1 ∈ ℕ ∧
taille$1 ≤ 2147483647 ∧
taille$1 = card(ens) ∧
taille$1 = 0 ⇒ ens = ∅ ∧
taille$1 = 1 ⇒ ens = {v1$1} ∧
taille$1 = 2 ⇒ ens = {v1$1,v2$1} ∧
taille$1 = 3 ⇒ ens = {v1$1,v2$1,v3$1} ∧
btrue ∧
"‘enter preconditions in previous components’" ∧
ee ∈ ℕ ∧
ee ≤ 2147483647 ∧
lll_1.enter.30 ∧
"‘enter preconditions in this component’" ∧
"‘Local hypotheses’" ∧
¬(taille$1 = 0) ∧
¬(taille$1 = 1) ∧
¬(taille$1 = 2) ∧
¬(v1$1 = ee) ∧
¬(v2$1 = ee) ∧
¬(v3$1 = ee) ∧
taille$1 = 3 ∧
card(ens ∪ {ee}) ≤ 3 ∧
"‘Check that the invariant (ov$1 = ov) is preserved by the operation,
ref 8’"
⇒
bfalse
```

As *taille$1* = *3*, we know that *ens* = {*v1$1,v2$1,v3$1*}. On the other hand, *taille$1* = card*(ens)* thus the three *vi$1* are different. But *ee* is different from each *vi$1*, thus card*(ens*∪{*ee*}*)* = *4*. It is the card*(ens*∪{*ee*}*)* ≤ *3* hypothesis that is contradictory. The initial goal displayed in the Interactive Prover contains the local hypotheses:

```
"‘Local hypotheses’" ∧
        ¬(taille$1 = 0) ∧
        ¬(taille$1 = 1) ∧
        ¬(taille$1 = 2) ∧
        ¬(v1$1 = ee) ∧
```

$\neg$(v2\$1 = ee) $\wedge$
$\neg$(v3\$1 = ee) $\wedge$
taille\$1 = 3 $\wedge$
card(ens $\cup$ {ee}) $\leq$ 3 $\wedge$
"'Check that the invariant (ov\$1 = ov) is preserved by the operation, ref 8'"
$\Rightarrow$
bfalse

There are two methods to load these hypotheses:

- the *Deduction* (dd) command: the hypotheses are then directly loaded without Prover action. But the Prover cannot reduce to the minimum the identifiers in use as it does for the other hypotheses.
- the *Prove* (pr) command: the hypotheses are then loaded by the Prover that links on the Proof. The initial goal is thus transformed.

In our case the **bfalse** case cannot be transformed. We can thus load the hypothesis with the *Prove* command, using the Red option to avoid an untimely triggering of proofs by cases.

```
PRI > pr(Red)
Starting Prover Call
```

The goal becomes:

bfalse

The card*(ens*$\cup${*ee*}*)* $\leq$ *3* contradictory hypothesis is transformed: it now appears under two different forms. Without even trying to understand these new formulations, we easily verify that they are still contradictory when card*(ens*$\cup${*ee*}*)*. We simply indicate one of the forms to the Prover using the *FalseHypothesis* command:

```
PRI > fh(0<=2-card(ens)+card(ens /\ {ee}))
Starting False Hypothesis
```

The goal becomes:

$\neg$(0 $\leq$ 2-card(ens)+card(ens $\cap$ {ee}))

This means that the proof now amounts to demonstrate the negation of this formula. For reasons of proof coherence, it is impossible to delete an hypothesis - this is why the formula is still in hypothesis. This is not a problem as the *Predica*te Prover is not used to demonstrate $\neg P$. We can attempt the proof:

```
PRI > pr
Starting Prover Call
```

The goal becomes:

0 $\leq$ -card({v1\$1} $\cap$ {v2\$1})-card({v1\$1,v2\$1} $\cap$ {v3\$1})
-card({v1\$1,v2\$1,v3\$1} $\cap$ {ee})

This is rather complex! We still have to help the Prover before starting this proof. Let us go back:

```
PRI > ba
```

The goal becomes:

$$\neg(0 \leq 2\text{-}\mathsf{card}(\text{ens})+\mathsf{card}(\text{ens} \cap \{\text{ee}\}))$$

We restart the proof in reduced mode to simplify the goal without proofs by cases or exploratory replacements. Thus we apply the method described in the 6.1 chapter exactly.

```
PRI > pr(Red)
Starting Prover Call
```

The goal becomes:

$$0 \leq \text{-}3+\mathsf{card}(\text{ens})\text{-}\mathsf{card}(\text{ens} \cap \{\text{ee}\})$$

How can we help the proof at this step? The $ens \cap \{ee\}$ expression stands for the empty set and such simplification in a goal is seldom beneficiary. We can try and attract the Prover attention to this:

```
PRI > ah(ens /\ {ee} = {})
Starting Add Hypothesis
```

The goal becomes:

$$\text{ens} \cap \{\text{ee}\} = \varnothing$$

Let us attempt to demonstrate this:

```
PRI > pr
Starting Prover Call
```

The goal becomes:

$$\text{ens} \cap \{\text{ee}\} = \varnothing \;\Rightarrow\; 0 \leq \text{-}3+\mathsf{card}(\text{ens})\text{-}\mathsf{card}(\text{ens} \cap \{\text{ee}\})$$

This new hypothesis proof has succeeded. Thanks to it the main proof could be successful:

```
PRI > pr
Starting Prover Call
```

The goal becomes:

$$0 \leq \text{-}\mathsf{card}(\{\text{v1\$1}\} \cap \{\text{v2\$1}\})\text{-}\mathsf{card}(\{\text{v1\$1,v2\$1}\} \cap \{\text{v3\$1}\})$$

This is not enough. Let us return to replace `pr` with `pr(Red)` according to the general method:

```
PRI > ba
```

The goal becomes:

$$\text{ens} \cap \{\text{ee}\} = \varnothing \;\Rightarrow\; 0 \leq \text{-}3+\mathsf{card}(\text{ens})\text{-}\mathsf{card}(\text{ens}\cap\{\text{ee}\})$$

```
PRI > pr(Red)
Starting Prover Call
```

The goal becomes:

$$3 \leq \mathsf{card}(\mathrm{ens})$$

This goal is evident since $\mathsf{card}(ens) = taille\$1$ and $taille\$1 = 3$. The Prover fails as it does not replace the $\mathsf{card}(ens)$ expression with $taille\$1$. Without inquiring why, we simply do it manually:

```
PRI > eh(card(ens),taille$1,Goal)
Starting use Equality in Hypothesis
```

The goal becomes:

$$3 \leq \mathrm{taille\$1}$$

Let us attempt the proof:

```
PRI > pr
Starting Prover Call
```

The initial goal reapers displayed in green; the proof is complete. The proof tree is as follows:

```
Force(0)
  pr(Red)
    fh(0<=2-card(ens)+card(ens /\ {ee}))
      pr(Red)
        ah(ens /\ {ee} = {})
          pr
          pr(Red)
            eh(card(ens),taille$1,Goal)
              pr
  Next
```

As we have seen, a proof by contradiction often consists in indicating the contradictory hypothesis to the Prover. In this example, we have additionaly performed two interventions on the hypothesis negation proof. This demonstration might appear complex for such a simple problem but it is performed more rapidly than a real manual formal demonstration.

8.2 Arithmetic proof with divisions

The example given here is a proof concerning the integer division and requiring to know the variation interval of the remainder of such a division. We place ourselves in the case where the Prover has no mathematical knowledge of the remainders. The proof will be done by adding user rules. We are to see how by using these functionalities we can reduce these rules to the minimum. Let us consider the following proof obligation:

```
"‘Included,imported and extended machines invariants’" ∧
c1$1 ∈ 0..150 ∧
c2$1 ∈ 0..150 ∧
nmesure$1 ∈ ℕ ∧
ntransfert$1 ∈ ℕ ∧
btrue ∧
0 ≤ c1$1 ∧
c1$1 ≤ 150 ∧
0 ≤ c2$1 ∧
c2$1 ≤ 150 ∧
"‘Previous components invariants’" ∧
c1$1 ∈ 0..120 ∧
c2$1 ∈ 0..120 ∧
c1$1-c2$1 ∈ -1..1 ∧
"‘Component invariant’" ∧
c2 = c2$1 ∧
c1 = c1$1 ∧
nmesure$1 = ntransfert$1 ∧
ncycle = ntransfert$1 ∧
c1$1 ≤ 120 ∧
c2$1 ≤ 120 ∧
0 ≤ 1+c1$1-c2$1 ∧
-1 ≤ c1$1-c2$1 ∧
0 ≤ 1-c1$1+c2$1 ∧
c1$1-c2$1 ≤ 1 ∧
equi_1.cycle.20 ∧
"‘Local hypotheses’" ∧
nc1 ∈ 0..150 ∧
nc2 ∈ 0..150 ∧
c1$1-nc1 ∈ -4..4 ∧
c2$1-nc2 ∈ -4..4 ∧
nc1+nc2 ≤ c1$1+c2$1 ∧
nc1-(nc1-nc2)/2 ∈ 0..150 ∧
nc2+(nc1-nc2)/2 ∈ 0..150 ∧
nmesure$1+1 ∈ ℕ ∧
ntransfert$1+1 ∈ ℕ ∧
"‘Check operation refinement, ref 11’"
⇒
nc1-(nc1-nc2)/2-(nc2+(nc1-nc2)/2) ∈ -1..1
```

It is easy to verify intuitively that this proof obligation is true. Indeed, if the present expression were a calculus in the real numbers set, we would have:

nc1-(nc1-nc2)/2-(nc2+(nc1-nc2)/2) = 0

If (nc1-nc2) can be divided by 2, the integer calculus is similar to the real calculus, it is thus given as 0. If (nc1-nc2) cannot be divided by 2, the considered expression is worth 1 or -1. Indeed this is not a demonstration, but a way to understand *why* this proof obligation is true. This is the 'intuitive demonstration' evoked in this document.

Let us start with the formal proof. After loading the proof obligation, the goal display area contains the following formula:

```
"‘Local hypotheses’" ∧
nc1 ∈ 0..150 ∧
nc2 ∈ 0..150 ∧
c1$1-nc1 ∈ -4..4 ∧
c2$1-nc2 ∈ -4..4 ∧
nc1+nc2 ≤ c1$1+c2$1 ∧
nc1-(nc1-nc2)/2 ∈ 0..150 ∧
nc2+(nc1-nc2)/2 ∈ 0..150 ∧
nmesure$1+1 ∈ ℕ ∧
ntransfert$1+1 ∈ ℕ ∧
"‘Check operation refinement, ref 11’"
⇒
nc1-(nc1-nc2)/2-(nc2+(nc1-nc2)/2) ∈ -1..1
```

Indeed the local hypotheses have not yet been loaded. The first thing to do is to start the proof to check where it stops, according to the method described in section 6.1. We have only to use the `pr` command and observe the goal that appears:

$$0 \leq 1\text{-nc1+nc2+}2\times((\text{nc1-nc2})/2)$$

We notice that the Prover tries to demonstrate that after simplification the expression is inferior or equal to 1. This is half the initial proof consisting in proving a membership to -1..1: there is therefore another goal to be demonstrated after, establishing that the initial expression is superior to -1.

We assume that the Prover contains no rules on the remainders of integer division (for example, we have seen this by examining the rule database with the *SearchRule* `sr`) command . Thus there is a mathematical knowledge to be added - the two parts of the proof would benefit from it. This is why it is better to return to the beginning of the demonstration before proceeding to this knowledge addition. This is what we do with the *Reset* (`re`) command.

The *a priory* minimum rules required to perform this demonstration are as follows:

```
THEORY IntDiv IS

  b*(a/b) == a - (a mod b);

  a: NATURAL &
  b: NATURAL &
  not(b = 0)
  =>
  (a mod b) : 0..(b-1);

  a: NATURAL &
  b<=0 &
  not(b = 0)
  =>
  (a mod b): 0..(1-b);

  a<=0 &
  b: NATURAL &
  not(b = 0)
  =>
  (a mod b): (1-b)..0;

  a<=0 &
  b<=0 &
  not(b = 0)
  =>
  (a mod b): (b-1)..0

END
```

We have given these rules in theory language in the format allowing us to write them in a *.pmm component* file. This format requires a few comments:

- To indicate that a variable must be a positive integer, we write $a \in \mathbb{N}$ instead of $a \in \mathbb{Z} \ \wedge \ 0 \leq a$ as $\mathbb{N}$ belongs to the Prover basic symbols, contrary to $\mathbb{Z}$.
- To indicate that a variable must be a negative integer, we simply write $a \leq 0$, without specifying $a \in \mathbb{Z}$. Indeed if a is not an integer, for example $a = \mathsf{TRUE}$, then the a mod b is ill-typed and could not appear in a proof obligation stemming from a component whose type control is correct. Thus we avoid the $\mathbb{Z}$ symbol.

The modulo we have here is the extension of $\mathbb{Z} \times \mathbb{Z}_1$ of the definition on $\mathbb{N} \times \mathbb{N}_1$:

"If a and$b$ are two natural integers and b non nil, then there one and only one (q, r) ordered pair so that $a = bq + r$ and$r < b$. By definition: $q = a/b$ and $r = a$ mod b."

This definition can be extended to $\mathbb{Z} \times \mathbb{Z}_1$ in a non ambiguous way so that $a = b \times (a/b) + (a \text{ mod } b)$ remains true and the sign simplification rules for a quotient are natural. This is the definition of the modulo that is used in Atelier B. It is built up from this equality and from the quotient membership rules - we have thus fully described it in `IntDiv`.

The first rule: `b*(a/b) == a - a mod b` does not have an indifferent form. We have selected a rewriting rule allowing us to eliminate a division by creating a modulo, thus allowing us to replace the divisions in the goal and hypotheses. Moreover, the selected writing is $b \times (a/b)$ instead of $(a/b) \times b$ because the arithmetic solver writes the simplest coefficients first. Writing such a rule directly in its optimum form is not easy and requires experience. It is nonetheless always possible to improve on the proof form when it proves non-practical.

We introduce these rules in a .pmm file from the concerned component then load this file using the *PmmCompile* (`pc`) command. How can we use this mathematical knowledge in our example?. For us a is worth $nc1 - nc2$ and$b$ is worth 2. b is thus positive. But we have to make two cases according to $nc1 - nc2$ sign; these cases have to be done as early as possible and in any case before the division of the goal into two sub-goals.

It would not be logical to make the two cases $nc1 - nc2 \leq 0$ and $nc1 - nc2 \geq 0$ from the current goal since $nc1$ and $nc2$ are specified in the local hypotheses. These are still in the goal, it is thus not possible to use them before their loading. We have two means of loading local hypotheses:

- the *Deduction* (`dd`) command: the hypotheses is then directly loaded without any Prover intervention. The Prover in particular cannot reduce the identifiers used at the minimum as it does with the other hypotheses.
- the *Prove* (`pr`) command: the hypotheses are then loaded by the Prover that links to the proof. The initial goes will then be divided en two as seen above.

To avoid making two cases ($nc1 - nc2 \leq 0$ and $nc1 - nc2 \geq 0$) for the two sub-goals, we must use `dd`. The `pr(Red)` option in the `pr` command does not avoid the creation in this case of two sub-goals - as it is not a Prover attempt to prove by cases but indeed a basic rule to prove $x \in u\,..\,v$: we must demonstrate $u \leq x$ then $x \leq v$. These considerations can be done by doing some quick tests with the `pr` from the previous goal. Thus it does not require the Prover inner programming.

Thus the first command we will use `dd`:

```
PRI > dd Starting Deduction
```

The goal becomes:

nc1-(nc1-nc2)/2-(nc2+(nc1-nc2)/2) ∈ -1..1

The local hypotheses now end the list displayed in the hypotheses window.

To make two cases according to $nc1 - nc2$ sign, we have the *DoCases* (`dc`) command. Let us recall the two forms of this command:

- `dc(P)`: enables to position in the P and $\neg caseP$;

- `dc(v,E)` : proof for v with each of E.

We must use the first form. In a proof by cases, it is always advisable to start with the most difficult case: here certainly the case where $nc1 - nc2$ is negative. Indeed proof on negative integers is often more difficult. We thus select $nc1 - nc2 \leq 0$ for P.

The reader will have noticed that we use the $\leq$ symbol, instead of $\geq$, $<$, $>$. In fact, $\leq$ is a Prover basic, the other symbols being brought back to it. In spite of the presence of an automatic normalization system, it is advised to use these basic symbols in priority. Their list is given in the Prover Reference Manual, [?].

```
PRI > dc(nc1-nc2<=0)
Starting Do Cases
```

The goal becomes:

$$\text{nc1-nc2} \leq 0 \;\Rightarrow\; \text{nc1-(nc1-nc2)/2-(nc2+(nc1-nc2)/2)} \in \text{-1..1}$$

This proof by cases hypothesis is a local hypothesis, it appears in the goal. The previous considerations on loading local hypotheses being still valid, we load this hypothesis with `dd`.

```
PRI > dd
Starting Deduction
```

The goal becomes:

$$\text{nc1-(nc1-nc2)/2-(nc2+(nc1-nc2)/2)} \in \text{-1..1}$$

Up till now we have not unloaded any goal, the proof tree is thus a consistent ascent. We can control this in the command line area (global situation window) whose display is as follows:

```
Force(0) &
  dd &
    dc(nc1-nc2<=0) &
      dd &
        Next
```

We are in the case where $nc1 - nc2$ is negative. We can now introduce the remainder variation interval division by 2 of $nc1 - nc2$ by using the added rules. We still ignore *how* this element will be used, but it is clearly required.

The simplest way to indicate this variation interval is to add an hypothesis: (nc1-nc2) `mod` 2 $\in$ -1..0. The Prover would not know how to prove this new hypothesis as we have seen it does not have the required rules. We shall thus do it with the `IntDiv` theory corresponding rule. This rule is:

```
a<=0 &
b: NATURAL &
not(b = 0)
=>
(a mod b) : (1-b)..0;
```

To be able to use this rule, the goal to be proven must have exactly the form of the rule consequent. We shall thus add the (nc1-nc2) mod 2 $\in$ 1-2. .0 non-simplified hypothesis that exactly corresponds with our rule. The required hypotheses will be done afterwards.

```
PRI > ah((nc1-nc2) mod 2: 1-2..0)
Starting Add Hypothesis
```

The goal becomes:

(nc1-nc2) mod 2 $\in$ 1-2. .0

We now have to apply our rule - which is the fourth in the `IntDiv` theory. For such rules with no rewriting, the *ApplyRule* (`ar`) command provides two modes: `Once` (applied once) and `Multi`apply as long as possible). Here, these two modes are equivalent as the rule does not re-apply to its antecedents.

```
PRI > ar(IntDiv.4,Once)
Starting Apply Rule
```

The goal becomes:

nc1-nc2 ≤ 0

It is really the first instanced antecedent of the rule Let us try to demonstrate it with a simple call to the Prover:

```
PRI > pr
Starting Prover Call
```

The goal becomes:

$2 \in \mathbb{N}$

This is the second antecedent; the previous goal has been unloaded. Let us proceed:

```
PRI > pr
Starting Prover Call
```

The goal becomes:

$\neg(2 = 0)$

This is the third antecedent, the previous goal has been unloaded. Let us proceed:

```
PRI > pr
Starting Prover Call
```

The goal becomes:

$$(\text{nc1-nc2})\ \texttt{mod}\ 2 \in 1\text{-}2..0 \ \Rightarrow\ \text{nc1-(nc1-nc2)/2-(nc2+(nc1-nc2)/2)} \in \text{-}1..1$$

Again we find the goal preceding the hypothesis addition, under the required hypothesis. This part of the demonstration benefited both from the added rule and the automatic proof functionalities - this allows us to waste no time on the parts that can be automatically demonstrated. The command line area contains the following proof tree:

```
Force(0) &
  dd &
    dc(nc1-nc2<=0) &
      dd &
        ah((nc1-nc2) mod 2: 1-2..0) &
          ar(IntDiv.4,Once) &
            pr &
            pr &
            pr &
          Next
```

The `Next` keyword, directly indented under the `ah` command indicates that the present goal is a sub-goal created by this command. Let us recall that `ah(H)` creates two sub-goals from a B goal: the H sub-goal then the $H \Rightarrow B$ one. We thus are on the second sub-goal as `Next` is the second indented keyword under `ah`, after `ar`.

The user rule was used to create a new hypothesis; it is thus a kind of front generation. It would have been possible to create a *forward* rule to get the same result but this would have been less simple. Using the addition of a hypothesis allows us to use a rule easily only when written according to mathematical considerations.

We have introduced the remainder variation interval but the Prover still cannot succeed without using the first rule in the `IntDiv` theory:

```
b*(a/b) == a - (a mod b);
```

If we use the `pr` command, it is evident that the current goal will not be unloaded. But it is nonetheless useful that the Prover simplifies the new hypothesis and starts the proof by simplifying the goal. To perform all this without starting exploratory proofs by cases, we can use `pr(Red)`:

```
PRI > pr(Red)
Starting Prover Call
```

The goal becomes:

$$0 \leq 1\text{+nc1-nc2-}2\times((\text{nc1-nc2})/2)$$

As expected, the Prover has simplified the goal and deleted the interval by producing two sub-goals. The hypothesis has indeed been simplified and loaded, it is displayed at the bottom of the hypotheses list. We must now use `IntDiv` first rule. It is a rewriting rule for which the *ApplyRule*(`ar`) command provides two modes: `Goal` (rewriting in the goal) and `Hyp(h)` (rewriting in the hypotheses). We apply it to the goal:

```
PRI > ar(IntDiv.1,Goal)
Starting Apply Rule
```

The goal becomes:

$$0 \leq 1\text{+nc1-nc2-(nc1-nc2-(nc1-nc2)}\ \texttt{mod}\ 2)$$

Now, we can suppose nothing is missing for the proof of this sub-goal to succeed. We can thus try to call the complete Prover:

```
PRI > pr
Starting Prover Call
```

The goal becomes:

$$\text{nc1-(nc1-nc2)/2-(nc2+(nc1-nc2)/2)} \leq 1$$

This is the second sub-goal required to demonstrate the membership to the $-1..1$ interval. We notice that this goal has not been simplified: the Prover has in fact stopped when the first sub-goal failed.

The proof tree is now as follows:

```
Force(0) &
  dd &
    dc(nc1-nc2<=0) &
      dd &
        ah((nc1-nc2) mod 2: 1-2..0) &
          ar(IntDiv.4,Once) &
            pr &
            pr &
            pr &
          pr(Red) &
            ar(IntDiv.1,Goal) &
              pr &
            Next
```

The `Next` sub-goal is the second to be indented under `pr(Red)`; this means it is the second sub-goal created by this command, the one corresponding to the interval second bound. As previously, we know that the proof of this goal will not succeed without using the `IntDiv` second rule. To perform its simplification, we must use `pr(Red)` instead of `pr` that would try harmful proofs by cases.

```
PRI > pr(Red)
Starting Prover Call
```

The goal becomes:

$$0 \leq \text{1-nc1+nc2+2}\times\text{((nc1-nc2)/2)}$$

As previously we use `IntDiv` first rule:

```
PRI > ar(IntDiv.1,Goal)
Starting Apply Rule
```

The goal becomes:

$$0 \leq \text{1-nc1+nc2+(nc1-nc2-(nc1-nc2) } \mathtt{mod}\text{ 2)}$$

Everything is ready and we can start the automatic proof:

```
PRI > pr
Starting Prover Call
```

The goal becomes:

$$\neg\text{(nc1-nc2} \leq 0) \;\Rightarrow\; \text{nc1-(nc1-nc2)/2-(nc2+(nc1-nc2)/2)} \in \text{-1..1}$$

This goal is the second proof case if $nc1 - nc2$ is positive. We have completed half of the main proof . The proof tree is displayed in the command line area:

```
Force(0) &
  dd &
    dc(nc1-nc2<=0) &
      dd &
        ah((nc1-nc2) mod 2: 1-2..0) &
          ar(IntDiv.4,Once) &
            pr &
            pr &
            pr &
          pr(Red) &
            ar(IntDiv.1,Goal) &
              pr &
            pr(Red) &
              ar(IntDiv.1,Goal) &
                pr &
      Next
```

The `Next` keyword is the second indented one immediately under `dc`. It is thus the second case of this proof by cases. The $\neg(nc1 - nc1 \leq 0)$ local hypothesis indicates that we now assume $nc1 - nc2$ to be positive.

We have returned just under the `ah` command we used to indicate the remainder variation interval. This variation interval is indeed no longer the same; thus we must specify it again for the case where $nc1 - nc2$ is positive. The adequate rule is:

```
a: NATURAL &
b: NATURAL &
not(b = 0)
=>
(a mod b) : 0..(b-1);
```

As previously, we must introduce this variation interval before the appearance of the two sub-goals. We first load the $\neg(nc1 - nc1 \leq 0)$ local hypothesis presently in the goal - as it is necessary to specify the remainder variation interval. Loading must be done without using the Prover:

```
PRI > dd
Starting Deduction
```

The goal becomes:

nc1-(nc1-nc2)/2-(nc2+(nc1-nc2)/2) ∈ -1..1

We proceed as with the first proof case:

```
PRI > ah((nc1-nc2) mod 2: 0..2-1)
Starting Add Hypothesis
```

The goal becomes:

(nc1-nc2) mod 2 ∈ 0..2-1

The adequate rule is the second in the `IntDiv` theory:

```
PRI > ar(IntDiv.2,Once)
Starting Apply Rule
```

The goal becomes:

nc1-nc2 ∈ ℕ

We naturally try and load this goal with a call (`pr`) to the Prover. Alas this proof fails in spite of the presence of the $\neg(nc1 - nc2 \leq 0)$ hypothesis. This hypothesis was evidently not used. It is an opportunity to use the method consisting in replaying hypotheses in the Prover (refer section 6.7.2).

```
PRI > ah(not(nc1-nc2<=0))
Starting Add Hypothesis
```

The goal becomes:

¬(nc1-nc2 ≤ 0) ⇒ nc1-nc2 ∈ ℕ

The hypothesis already exists as such: we do not have to repeat its demonstration. We restart the proof:

```
PRI > pr
Starting Prover Call
```

The goal becomes:

$$2 \in \mathbb{N}$$

It is exactly the next antecedent of the `IntDiv` theory second rule. The simple fact of replaying the key hypothesis in the Prover has enabled it to demonstrate the previous goal. Let us proceed:

```
PRI > pr
Starting Prover Call
```

The goal becomes:

$$\neg(2 = 0)$$

Let us proceed:

```
PRI > pr
Starting Prover Call
```

The goal becomes:

$$(\text{nc1-nc2}) \ \mathsf{mod}\ 2 \in 0..2\text{-}1 \ \Rightarrow\ \text{nc1-(nc1-nc2)/2-(nc2+(nc1-nc2)/2)} \in \text{-1..1}$$

We have thus added the hypothesis specifying the remainder variation interval. We have now to apply the `(a/b) == a - (a mod b)` rule after simplifying the goal:

```
PRI > pr(Red)
Starting Prover Call
```

The goal becomes:

$$0 \leq \text{1+nc1-nc2-2}\times\text{((nc1-nc2)/2)}$$

then:

```
PRI > ar(IntDiv.1,Goal)
Starting Apply Rule
```

The goal becomes:

$$0 \leq \text{1+nc1-nc2-(nc1-nc2-(nc1-nc2)}\ \mathsf{mod}\ 2)$$

We can try and unload this goal:

```
PRI > pr
Starting Prover Call
```

The goal becomes:

$$\text{nc1-(nc1-nc2)/2-(nc2+(nc1-nc2)/2)} \leq 1$$

The displayed goal refers to the second bound interval ; we shall process it similarly:

```
PRI > pr(Red)
Starting Prover Call
```

The goal becomes:

$$0 \leq \text{1-nc1+nc2+2}\times\text{((nc1-nc2)/2)}$$

Then

```
PRI > ar(IntDiv.1,Goal)
Starting Apply Rule
```

The goal becomes:

$$0 \leq \text{1-nc1+nc2+(nc1-nc2-(nc1-nc2) mod 2)}$$

We can try and prove this last sub-goal:

```
PRI > pr
Starting Prover Call
```

As the last sub-goal has disappeared, the proof obligation initial goal is now displayed in green: the proof is completed. We simply quit the proof to save the demonstration. The final proof tree is as follows:

```
Force(0) &
  dd &
    dc(nc1-nc2<=0) &
      dd &
        ah((nc1-nc2) mod 2: 1-2..0) &
          ar(IntDiv.4,Once) &
            pr &
            pr &
            pr &
          pr(Red) &
            ar(IntDiv.1,Goal) &
              pr &
            pr(Red) &
              ar(IntDiv.1,Goal) &
                pr &
      dd &
        ah((nc1-nc2) mod 2: 0..2-1) &
          ar(IntDiv.2,Once) &
            ah(not(nc1-nc2<=0)) &
              pr &
            pr &
            pr &
          pr(Red) &
            ar(IntDiv.1,Goal) &
              pr &
            pr(Red) &
              ar(IntDiv.1,Goal) &
                pr &
  Next
```

The two main proof cases can easily be identified in this tree indentation.

To perform this demonstration, we have used only five rules written in nearly fully mathematical format. These rules are almost definitions. They will easily be validated and might be re-usable. We have avoided using too specific rules thanks to the Interactive Prover functionalities. It is this principle of using simple rules with the Prover advanced functionalities that enables us to employ only a minimum of added rules. Thus we avoid rebuilding an Automatic Prover dedicated to our proof from complex user rules.

Another important remark: we did not do the whole demonstration manually. We concentrated on the tricky parts by unloading all easy goals with the `pr` command. Such a demonstration is both easier and safer than a manual demonstration as the latter is often tiresome when we forbid ourselves to use intuitive shortcuts that cannot be accepted in a formal proof.

Chapter 9

Frequently asked questions

9.1 Pr may mislead us

There is a fundamental difference between the `mp` command and the `pr` command. Regardless of the prover force used, the `mp` command applies rules of goal and hypotheses simplification, and some resolution mechanisms.
The `pr` command is built according to the same principles, but contains other mechanisms. Especially, some simplification heuristics of the existential predicates, of the last equalities loaded in hypotheses and processing the starting up of proof by cases that improve the efficiency of this command.

For example, if the current goal has the following form

$x \in S$

and the hypothesis

$x \in A \cup a$

does exist, then the current proof is divided in two cases:

$x = a \;\Rightarrow\; a \in S \;\wedge$
$x \in A \;\wedge\; not(x = a) \;\Rightarrow\; x \in S$

Another example, if the $dom(A * B)$ formula appears in the current goal P, then two cases are generated:

$(B = \varnothing \;\Rightarrow\; [dom(A * B) := \varnothing]P) \;\wedge$
$(not(B = \varnothing) \;\Rightarrow\; [dom(A * B) := A]P)$

$[f := g]P$ here means that all the f occurencies in P are replaced with g.

However, the proof by cases can be useless and require to prove a lemma as many times as there are cases. Indeed, these proof by cases are started up according to some local criteria that are not relevant.

For example, if the lemma is demonstrated by contradiction (several hypotheses are contradictory), the starting up of a proof by cases will needlessly multiply the number of lemma to demonstrate by contradiction.

It is advised to try in interactive proof the `mp` command before the `pr` command. If `mp` fails, then we check if `pr` enables to conclude or to change the goal in a more easy provable form.

9.2 Use of a proof plan

Writting a proof plan before starting the interactive demonstration is required if we want this proof to be as small and productive as possible.

It is first advised to make sure that the proof obligation to be demonstrated is true. This is done by inspecting the goal and the local hypotheses. During this inspection, you should be able to determine the elements leading to the demonstration of this proof obligation, that is to say which are the required hypotheses. If necessary, these elements will be written on paper in order to be easily found again. If the proof obligation is complex and difficult to read, it is advised to use the formula logic analyser in order to have a more synthetical view of the proof obligation. If, on the opposite, the proof obligation seems to be false, a counter-example must be shown, that is to say a peculiar valuation of the variables defined in hypothesis which enables to demonstrate the contrapositive of the current goal.

The proof plan may contain:

- the type of demonstration to realize (proof by cases, proof by contradiction, decomposition of the goal). The determination of the demonstration type is established by the user according to the result of the proof obligation inspection, and according to his experience concerning the Atelier B interactive proof.
- the ordered list of the hypotheses to be added. This list can be filled in with the interactive commands used for the demonstration of each hypothesis.
- the current step of the proof tree.

This proof plan enables to:

- locate oneself in the proof tree: know what is being proved.
- determine the path covered and what is left.

- minimize the interactive demonstration size. For example, it is better to add a hypothesis before starting up a proof by cases rather than adding it successively for each of its cases.
- reuse demonstrations parts from other proof obligations, if for example, the same hypotheses are added.

Don't forget to try a *TryEveryWhere* (`te` command) as soon as you think that the demonstration you have just performed can be successfully applied to other proof obligation of the operation or of the component.

9.3 Should a user rule be added?

It is not always easy to know if at a given step of the demonstration it is better to add a mathematical rule in order to demonstrate (or help to demonstrate) the current goal, or if it is better to go on using other commands of proof.

Several aspects have to be considered, and it will be up to the user to decide according to the B development context.

9.3.1 Rule validation

A mathematical rule added by the user has to be validated. Two cases can occur:

- the rule is automatically demonstrated by rules proof tools. The validation work is reduced to its simpliest expression. In that case, the rule can be added to the Pmm file and then be used.
- the rule is not automatically demonstrated. That does not mean that the rule is false but that the predicates prover and the arithmetic prover simply didn't succeed in demonstrating the rule. It can be due to the fact that some B operators are not handled in an efficient manner by the predicates prover (for example, closure), or due to the fact that the heuristics used for the proof are not enough efficient in this case.

  The rule must then be manually demonstrated. Some demonstration elements must be given by the rule creator in order to trace the reasoning used for its creation. These demonstration elements would enable a reader to ensure of the rule exactness, by refering to the B-Book axiomatic. One can be more demanding and write a full mathemathic demonstration of the rule exactness, by refering to the B-Book axiomatic.

  In all cases, a re-reading by a third party is required as according to our experience, it is very easy to create a false rule without being aware of it.

9.3.2 Lemmas simplification

An “obvious” goal may not be demonstrated by the predicates prover as the terms that it contains generate complex predicates during the goal translation into manipulable predicates by the predicates prover. In that case, it is sufficient to replace each complex term with a variable in order to obtain an easier demonstrable rule. Once this rule is demonstrated, it is added to the Pmm file, then used to demonstrate the goal. For example the goal,

$$\{xx | xx \in INT \ \wedge \ xx \mathsf{mod} 10 = 0\} \ \{xx | xx : \mathbb{Z} \ \wedge \ \exists yy.(yy \in \mathbb{Z} \ \wedge \ 10 * yy = xx)\} \ \wedge$$
$$\{xx | xx \in \mathbb{Z} \ \wedge \ \exists yy.(yy \in \mathbb{Z} \ \wedge \ 10 * yy = xx)\} \subseteq \{xx | xx \in \mathbb{Z} \ \wedge \ xx \mathsf{mod} 10 = 0\}$$
$$\Rightarrow$$
$$\mathsf{min}(\{xx | xx \in \mathbb{Z} \ \wedge \ xx \mathsf{mod} 10 = 0\}) = \mathsf{min}(\{xx | xx \in \mathbb{Z} \ \wedge \ \exists yy.(yy \in \mathbb{Z} \ \wedge \ 10 * yy = xx)\})$$

is not demonstrated by the predicates prover, neither by the prover.
On the opposite, the rule

$$a \subseteq b \ \wedge$$
$$b \subseteq a$$
$$\Rightarrow$$
$$\mathsf{min}(a) = \mathsf{min}(b)$$

is demonstrated by the rules proof tools.

```
PRI > vr(Back, (a<:b & b<:a => min(a)=min(b)))
The rule was proved
```

it is then added to the Pmm file, that then contains

```
THEORY MyRule IS
    a<:b &
    b<:a
    =>
    min(a)=min(b)
END
```

It is then loaded in memory using the `pc` command.

```
PRI> pc
Loading theory MYRule
```

It finally can be applied:

```
PRI> ar(MyRule.1,Once)
Starting Apply Rule
```

9.4 The various levels of interactive commands

The carrying on interactive commands can be divided in two categories:

- $\boxed{\text{high level commands}}$: `pp`, `pr`, `mp` that correspond to the performing of several rules and mechanisms. These are termination commands, which means that their application is required to demonstrate any goal, even btrue.
- $\boxed{\text{low level commands}}$: `dd`, `ah`, `ar`. These are commands performed step by step. There is no resolution. Their combination enables to carry on the proof. On the opposite of the commands given below, theses commands require a well knowledge/experience of the interactive prover.

9.5 Use of SearchRule

The SearchRule command can be used in two different situations:

- when the automatic prover doesn't succeed in demonstrating the current goal. The search for the rules that may be applied lets you to determine what is the decisive rule and the reason why this rule is not automatically applied. For example, a hypothesis can be missing. In this case, after adding this hypothesis the prover should apply this rule.
  Keep in mind that the SearchRule command displays all the rules likely to be applied and not the ones that are applied.
- when adding manual rules. The SearchRule command lets you check that the rule loaded in memory does correspond to the expected rule. Don't forget that the rules contained in a Pmm file are normalized during their loading.

9.6 Addition of false hypotheses

When adding a hypothesis, be careful not to add a false hypothesis as the demonstration of the hypothesis would then be impossible to do.

We know that, if P is the current goal and the `ah(H)` command is applied, then the goal becomes

$$H \ \wedge \ (H \ \Rightarrow \ P).$$

The error may consist in misspelling the name of a B identifier, or in inversing two variables, or in missparenthesising an expression: the error risks are numerous.

The prover doesn't indicate any anomaly and the user won't be warned of his error. Only an attentive reading can show him where the error occured. Sometimes, if several proof commands are entered after an AddHypothesis it may be very diffficult to find the error.

So please be careful when adding a hypothesis.

9.7 Number of required proof steps

Sometimes if a proof is too long we think that we've made an error and then we do it again whereas it is true. Generally, the proof are small but some can be very long.

We consider that 10 proof steps in average for the non automatically demonstrated PO is a good metric.

However, according to the model type and the way to modelize, it is possible to obtain complex POs, requiring properties addition and/or proof by cases. In this case, it is obvious that 10 steps is not a good value as it is underestimated.

For some B developments, it happens that some demonstrations have more than 500 steps, due to proof by cases, containing for example each of the numerous interactive commands almost indentic from one case to another (we can worry about the the conservation of this proof when models change).

A way to reduce a demonstration size is to simplify the B model. The other way consists in adding one or more user rules. Warning ! They must be easily checked and not too specific.

9.8 Rule that can not be applied

When using rules via the `ar` command, remember that if the goal has the following form $A \Rightarrow B$, A is not yet part of the hypotheses (perform a `dd` to do this).

So a rule with the following form

$$\mathsf{binhyp}(A) \Rightarrow B$$

cannot be applied as long as A is not a hypothesis.

Be careful to avoid using looping rules.

For example, the rules:

$$\begin{array}{l} bcall1(BackwardRule(rule.1)) \ \wedge \\ \mathsf{binhyp}(H) \ \wedge \\ (H \ \Rightarrow \ B) \\ \Rightarrow \\ B \end{array}$$

$$\begin{array}{l} bcall1(BackwardRule(rule.2)) \ \wedge \\ \mathsf{binhyp}(H) \ \wedge \\ \mathsf{bgoal}(C \ \Rightarrow \ D) \ \wedge \\ (H \ \Rightarrow \ B) \ \wedge \\ \Rightarrow \\ B \end{array}$$

can be applied infinitely.
It is possible to avoid this phenomenon:

- by applying rules in a unitary manner: `ar(rule.1, Once)` instead of `ar(rule)`
- by guarding the rules. The addition of the guard `bnot(bsearch(H,C,R))` avoids a loop by the `ar` command. On the opposite, a call to `pr(Tac(rule))` induces a loop.

In general, it is advised to avoid adding hypotheses in that way. It is better to use forward rules.

9.9 Use of pp(rp.0)

Using the predicates prover on the current goal whithout any hypothesis (command `pp(rp.0))` is very efficient, to demonstrate intermediaries lemmas. Indeed, `pp` is especially efficient when the hypotheses number is not very high. We can consider that its efficiency decreases in an exponantial way when the hypotheses number increases.

So, we merely select all the hypotheses needed to demonstrate the goal, using the `ah` command, then we apply the `pp(rp.0)` command.

PP is used for:

- arithmetic lemmas, (except ** and modulo),
- propositions,
- sets,
- quantified predicates.

Some symbols are not handled by `pp`. For instance: $closure$, $closure1$, $INTER$, $UNION$, inter, union. If your goal contains one of these symbols, it is advised to try the simplification/demonstration of the goal by the combination of manual rules/ `pr` command.

9.10 What is the difference between pr, pp and ap?

In interactive proof, there are 3 provers that each have their specificity:

- the automatic prover (called by mp or pr), that applies simplification/resolution global mechanisms and the rules base.
- the predicates prover that changes the goal to demonstrate in predicates, then perform some heuristics which have to demonstrate the lemma by contradiction.
- the arithmetic prover that runs only with an inequality in goal. It then tries to demonstrate the goal by contradiction via linear combination of the arithmetic hypotheses.

These three provers have partially common domains. It is advised to test one domain or the other one on a goal to see if it is indeed demonstrated.

The two last provers are only solvers indicating if the goal is demonstrated ornot, without transforming/simplifying it.

The automatic prover, applying equivalence rules, changes (simplification/dividing) the goal and can demonstrate it when it is simplified in btrue.